IN-62
26652
P-61

NASA Contractor Report 4381

Verification of the FtCayuga Fault-Tolerant Microprocessor System

Volume 1: A Case Study in Theorem Prover-Based Verification

Mandayam Srivas and Mark Bickford

CONTRACT NAS1-18972
JULY 1991

NASA Contractor Report 4381

Verification of the FtCayuga Fault-Tolerant Microprocessor System

Volume 1: A Case Study in Theorem Prover-Based Verification

Mandayam Srivas and Mark Bickford
ORA Corporation
Ithaca, New York

Prepared for
Langley Research Center
under Contract NAS1-18972

National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Program
1991

Contents

List of Figures

1 Introduction

NASA Langley Research Center has recently initiated a research effort to develop a validation methodology for life-critical digital (fly-by-wire) flight control systems. Such systems must meet stringent reliability requirements. The systems are expected to have a probability of failure as low as 10^{-9} for a 10 hour mission. Hence, as has been well-argued in [10], the design and validation methods employed for such systems must meet high standards. The designs must use fault-tolerant strategies to enable the continued operation of the system in the presence of component failures. The validation methods must ensure that there are no design errors in the system.

The process of formal verification is capable of showing that a system design satisfies a specified property for all inputs and initial conditions of the design. Hence it is a promising candidate for consideration as a validation technology for flight control systems. Digital control systems are implemented using a combination of hardware and software components. Hence, a formal verification technology for flight control systems must be applicable to both software and hardware designs. This work is concerned with the formal verification of a hardware design.

The paper presents a design and formal verification of a hardware system for a task that is an important component of a fault-tolerant computer architecture for flight control systems. The hardware system implements an algorithm for attaining *interactive consistency* (*byzantine agreement*) [6] among four microprocessors as a special instruction on the processors. The property verified ensures that an execution of the special instruction by the processors correctly accomplishes interactive consistency, provided certain preconditions hold. An assumption is made that the processors execute synchronously. For verification, we used a computer-aided hardware design verification tool, Spectool [7], and the theorem prover, Clio [2], both of which were developed at ORA. The microprocessor used in the system is called FtCayuga, which we designed by extending the formally verified microprocessor MiniCayuga [8].

A major contribution of the work is the demonstration of a significant fault-tolerant hardware design that is mechanically verified by a theorem prover. The work illustrates the advantage of using hierarchy and abstraction in system design specification to manage the complexity of formal verification. The work also demonstrates the value of a special-purpose tool that tailors the use of a theorem prover to a hardware domain in order to reduce the

effort required for formal verification.

This volume of the paper contains an overview of the work and an informal description of the specification and the proof of correctness. This volume gives only a part of the formalization of the system and the correctness theorems. The companion volume [3] of the paper contains the entire specification and all the correctness theorems we proved.

2 Formal Verification of Hardware Designs

Hardware verification involves the use of a formal verification system, such as a theorem prover, for analyzing properties of hardware circuit designs. Hardware verification can be used to show that a model of a circuit design satisfies a given functional or timing property for all valid inputs and initial conditions of the circuit. Thus, unlike the traditional method of circuit validation by design simulation, where the circuit is simulated only on a selected set of inputs, formal verification ensures complete test coverage for the property verified. The past few years have seen the successful completion of formal verification of several complex hardware designs ([8], [5], [4]), which shows that the technology is applicable to industry-scale problems. Formal design verification using a theorem prover consists of the following steps:

1. Specifying a model of the circuit design in the language or logic of the theorem prover.
2. Expressing the property to be verified about the design as a formula (*verification condition*) in the logic of the prover. The formula is usually expressed as an assertion expected to be true of all possible valid inputs and initial conditions of the circuit.
3. Proving that the verification condition is a logical consequence of the circuit specification. This step is performed using the theorem proving facilities supported by the prover.

While formal verification is powerful, in principle, it still remains a sophisticated and tedious process. A high level of specialized expertise is required to construct proofs using a theorem prover. A substantial part of the effort expended in verifying a design can often be reused to verify other designs. But, existing support facilities for transferring reuseable parts of specifications and proofs from one design to another are poor. One way to reduce

the tedium and sophistication involved is to build an integrated suite of tools that aid the user in the verification process. Developing general-purpose tools to support the verification process is hard because the techniques involved vary with the domain of the application. If the domain can be suitably restricted, then it is possible to automate or at least provide interactive computer assistance for a substantial part of the verification process.

Spectool is a computer-aided verification tool that we have built at ORA to study the above thesis. Spectool is intended to assist in using the Clio theorem prover to formally verify a class of hardware designs. In the present effort, we used Spectool to automatically generate formal specifications for various parts of our system and to streamline the construction of proofs of properties about the specifications. We constructed the proofs using Clio. The two tools used in the effort are summarized below.

2.1 Clio Summary

Clio is a system for proving properties about programs written in an executable functional language, Caliban. Caliban is a higher order, polymorphic, lazy functional language similar to Miranda[1] [9]. A property to be proved is expressed in the Clio assertion language as an arbitrary first order predicate calculus formula built from atomic literals. An atomic literal is an equation on Caliban expressions. In the present application, the specifications of the system and its components are expressed in Caliban. The verification conditions to be proved are expressed in the assertion language.

The basic proof technique of the Clio prover is *normalization*, i.e., simplifying expressions on the two sides of an equation to a common form to prove the equation. The prover supports a set of *proof tactics* that are useful in conjunction with normalization to prove more complex formulae. Some of the proof tactics available are: *case analysis*, *structural induction*, *fixpoint induction*, and *proof by contradiction*. The user can use the prover in interactive or automatic mode. In the interactive mode, Clio prompts the user with the available choices in proof tactics, and the user must make the appropriate selection until Clio proves the formula or discovers a contradiction. In the automatic mode, it makes the selection on its own based on a built-in strategy.

[1]Miranda is a trademark of Research Software Limited.

2.2 Spectool Summary

Spectool is a computer-aided verification tool targeted for synchronous hardware designs described at a level of representation that is comparable to the register-transfer level of hardware. The hardware representation model used by Spectool is described in detail in section 7. We designed Spectool based on our experience in using Clio to verify several hardware designs, the largest of which was the MiniCayuga processor design. The tool is aimed to reduce the effort required for verifying a design in the targeted class by automating most of the routine, but cumbersome, parts of the verification process. Generic parts of the technology are identified and encapsulated so that the user can reuse them by appropriate instantiation.

Spectool provides a graphical user-interface that the designer uses to draw a circuit diagram of a hardware design on an electronic canvas. The designer then annotates the circuit diagram with various pieces of useful information; for example, one piece of information is a set of conditions that are expected to hold at specified points during the operation of the circuit. The tool uses the annotated circuit diagram to automatically generate specification and verification conditions for the circuit. Spectool aids construction of the proof for the conditions by generating a set of control annotations to the prover. The control annotations assist in constructing a proof for the verification conditions either automatically or by applying a standard proof strategy. The tool provides a facility that the user can use to display the results of the prover in terms of the symbols and concepts introduced by the user at the circuit diagram level. This facility is helpful if the standard proof strategy does not succeed.

3 The Interactive Consistency Problem

Fault-tolerant architectures use replicated hardware resources to enable continued operation in the presence of physical failures of components. Whenever a single source data, such as inputs from a sensor, must be distributed to replicated resources, the distribution must be reliable, so that every replicated resource gets the same value. To come to a consensus about the value of the single source data, the replicated resources must use some kind of voting. The voting hardware must itself be distributed since the goal is to eliminate all single-point failures in the system. This problem of arriving at a consensus among a set of replicated resources has been called interactive consistency or Byzantine Generals problem.

The interactive consistency (IC) problem [6] in its most general form is defined as follows: Consider a set of n processors, of which no more than m are faulty. It is not known which of the processors are faulty. Suppose that each processor p has some private value of information $prvt_p$ (such as its clock value or its reading of a sensor). The problem is to devise an algorithm (for given m, $n \geq 0$) using messages where each nonfaulty processor p computes a vector of values, such that the following conditions hold:

1. *Unanimous* : the nonfaulty processors must compute the same value for every processor
2. *Correct* : the element of this vector corresponding to a given nonfaulty processor is the private value of that processor.

Many algorithms have been developed to perform this task. One of the main concerns of fault-tolerant system design is to determine how these algorithms can be incorporated into a computer system architecture. This paper presents a hardware design solution for the IC problem for the case in which there are four processing units; the hardware design is based on the algorithm described for the problem in [6]. This paper presents a formal verification that the hardware design implements the algorithm correctly. As shown in [6], there must be at least $3m + 1$ processors (replicated resources) to tolerate m byzantine faults in a system. Thus, our design can tolerate at most one faulty replicated region.

For the four processor case, the algorithm is as follows. The procedure consists of an exchange of messages, followed by the computation of the interactive consistency vector on the basis of the results of the exchange. Two rounds of message exchange are required. In the first round, the processors exchange their private values. In the second round, they exchange the results obtained in the first round. A faulty processor may "lie" or refuse to send messages. If a nonfaulty processor p fails to receive a message from some other processor, p simply chooses a value at random.

After the second round of exchange, each nonfaulty processor p records its private value for the element of the IC vector corresponding to p itself. The element corresponding to every other processor q is obtained by examining the three received reports of q's value. If at least two of the three reports agree, the majority value is used. Otherwise, a default value is used.

4 A Hardware Design for Interactive Consistency

In developing a hardware implementation for the algorithm we had to consider several design issues. These issues and the design decisions we made are described below.

We designed the system as a synchronous digital circuit. In this paradigm, the components of the design operate in lock-step, with parallel activities in the design synchronized with respect to an implicit global clock. In an actual hardware realization of our design, distinct components in the design may use distinct physical clocks. In practice, components that belong to two different fault regions, e.g., the processors, must use separate clocks since the fault regions must be electrically isolated from each other to preserve failure independence. In such a situation, the clocks must be synchronized so that the skew between any pair is bounded by a fixed amount. The result of the formal verification guarantees that for any given bound on the clock skew there exists a choice of clock frequency for which our design operates correctly.

Our design supports the IC operation as a machine instruction on the processors. This choice allows the software executing on the processors control over the IC operation. That is, a program executing on the processor determines when an IC computation is initiated, on what data the computation is performed, and how the result will be used. For the IC computation to yield a correct result, all the nonfaulty processors in the system must be synchronized so that they execute the IC instruction simultaneously. Once initiated, the IC computation will be completed after a fixed number (12) of clock cycles.

The processors in the system are identical copies of a processor called FtCayuga ("Fault-Tolerant Cayuga"). The FtCayuga design is an extension of the design of the formally verified processor MiniCayuga [8]. The extra hardware in FtCayuga provides the facilities needed to set up, initiate, and store the result of an interactive consistency operation. A major part of the interactive consistency algorithm is implemented outside of the processor in a separate piece of hardware called the voter. A separate voter is associated with each FtCayuga processor. Each processor sends its private value to its own voter as well as the other voters, which accomplishes the first round of exchange. The voters perform the rest of the computation, namely the second round of exchange and the voting. This implementation, with the tasks of exchanging data and voting on separate pieces of hardware, makes the design modular and the verification task simpler by decomposing the proof.

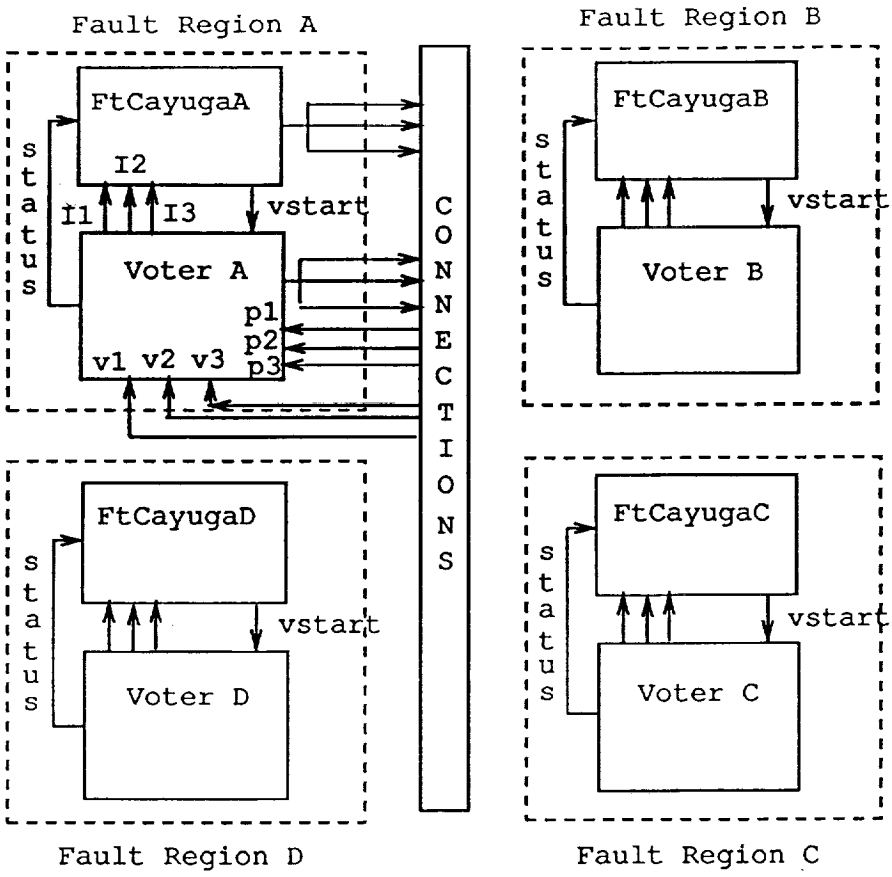


Figure 1: The Four Processor Architecture

FtCayuga accepts a single type of (asynchronous) interrupt signal that is intended to interrupt the normal sequence of execution of the processor. The interrupt signal, being a single source data, must itself be subjected to a byzantine vote. The topic of designing an appropriate interrupt mechanism for a fault-tolerant computer system is beyond the scope of this work. Our system is designed so that once an IC computation is initiated by the processors an occurrence of interrupts does not interfere with the IC computation.

For the purpose of our analysis, a voter-processor pair forms a single fault region. A fault region includes all the wires that are driven by the processor and the voter in that region. Thus, a failure of the voter, the processor, or any of the wires in a region is considered a failure of that region. The design achieves interactive consistency in the presence of at most one faulty region.

Figure 1 shows a top-level architecture of the hardware system. The dashed boxes identify the four fault regions. (The set of wires leading out of and into a region is shown

only for Fault Region A; the wires are identical for the other regions.) Each pair of connected components in the design are connected in a "point-to-point" fashion. That is, shared (buss) connections are deliberately avoided to isolate the fault regions electrically. Once an IC operation is initiated, the required data exchanges between components happen synchronously, i.e., without any handshake between the components. A "receiver" component reads its input assuming that the "sender" has provided the required data on the wire connected to the input. (If the sender fails to output a piece of data at the right time, the receiver reads in whatever value exists at the input for that data.) A voter outputs to its processor a consensus value corresponding to every processor outside the region and a (4-bit) `status` vector. The most significant bit of the `status` vector indicates if the voter is idle, i.e., ready to vote on a new set of values. The remaining bits of the `status` vector indicate whether the voter was able to determine a majority for a each of the processors outside the region. Having the `status` vector as an output of the voter eliminates the need to explicitly represent the default value used in the algorithm.

Every distinct unit of data, except the status vector, that is exchanged between two components in the design is a processor *word*. In the design and the specifications, we make no assumption about a data word other than that it contains a sequence of four *bytes*. For example, we make no assumption about the size of a word since the logic of the design, at least at the level at which it is being considered here, is not dependent on that information. The correctness proof developed here holds for words of arbitrary size. We can specialize our design specifications to a specific word size without redoing the entire proof of correctness. Such a specialization must be done to translate our design into a lower level of hardware representation.

Our aim is to design the system so that the voters (and processors) in the network are identical to each other. Consequently, the logic of a voter/processor cannot maintain any absolute index for the voters/processors in the network. A voter/processor is designed assuming that the there is a way to index the fault regions by means of a cyclic index set. (The voter and the processor in a region take the index associated with the region.) An index set, `INDEX`, is cyclic if there exists a successor (mod 4) function on the set such that the following conditions hold for every $ind \in$ `INDEX`:

$$\text{successor}^4(ind) = ind$$
$$\text{successor}^i(ind) = \text{successor}^{i-1}(\text{successor}(ind)),\ 1 \leq i \leq 3$$

Thus, for every voter/processor in a region with index *ind* there exists a voter/processor with indices successor(ind), successor^2(ind) and successor^3(ind). A component inside a fault region is designed assuming a fixed association between an input port and the index of the component connected to the port. For example, every voter expects to read at its input v*i* (or p*i*), data from its successor^i voter (or processor), for $1 \leq i \leq 3$. Every processor expects to read in at I*i* the component of the IC vector corresponding to its successor^i processor, for $1 \leq i \leq 3$.

The voters and the processors must be connected so that every voter/processor pair in a region can be assigned a unique index from a cyclic index set. For example, suppose the p1, p2, and p3 inputs of Voter A (in Figure 1) are connected to FtCayugaB, FtCayugaC, and FtCayugaD, respectively. This implies that B, C, and D are successor, successor^2 and successor^3 of A, respectively. Then, the inputs p1, p2 and p3 of Voter B, for example, must be connected to the FtCayuga's in regions C,D and A, respectively.

4.1 The Processor FtCayuga

FtCayuga, like its predecessor, MiniCayuga, is a 3-stage instruction pipelined RISC processor with thirty two general-purpose program registers. The processor normally takes three cycles to complete the three stages— *fetch*, *compute*, and *writeback*—involved in the execution of an instruction. Since the processor is pipelined to simultaneously execute three instructions in a cycle most of the time, it can attain an execution rate close to one instruction per cycle.

To support interactive consistency, we added the following new features to FtCayuga. A special register file (SREG) holds all the data relevant to IC computation. SREG[0] holds the value on which IC computation is performed. SREG[1] through SREG[3] contain, in order, the consensus values computed for the successor, successor^2 and successor^3 processors. SREG[4] contains the status vector upon completion of the IC computation. The processor provides three new instructions to help manage an IC computation. SADD and MOVE move data between REGFILE and SREG. ICOP initiates an IC computation. Once initiated, the processor performs IC computation on the current contents of SREG[0], provided SREG[0] is not changed by the program for one cycle immediately after ICOP is executed, and automatically stores the results from the voter at the proper destinations. Brief descriptions of these instructions follow.

```
SADD Reg Sreg   moves content of the general purpose register Reg into Sreg
                (acts like nop if destination is SREG[1] through SREG[4])
ICOP            clears status register (Sreg[4]) in sreg; initiates IC
                computation on SREG[0], provided SREG[0] is not changed once
                ICOP is initiated; nop if another ICOP is underway
MOVE Sreg Reg   moves content of the special register Sreg to the general
                purpose register Reg
```

The following is a FtCayuga "assembly level" program that one can use to perform an IC computation. In this program, the processor idles (busy waits) while the voter performs the 12-cycle IC computation. It is possible, however, to put these machine cycles to more productive use by performing some other necessary computation while waiting for the results from the voter. (The text following the characters || are comments.)

```
|| check if voter is free
Notfree     MOVE SREG4 REG1     || SREG4 is STATUS
            JIF REG1 Notfree    || jump if msb[REG1] is False
            NOP                 || pad nop to counter prefetching of pipeline
            SADD REG2 SREG0     || REG2 contains the byzantine data
            ICOP
|| check if IC computation is complete
Notready    MOVE SREG4 REG1
            JIF REG1 Notready
            NOP
|| move the results of IC vector to general register file
            MOVE SREG1 REG3
            MOVE SREG2 REG4
            MOVE SREG3 REG5
```

4.2 The Voter

In designing the voter, we needed to decide whether the different units of data read by the voter and the bits in a unit must be read serially or in parallel. The decision poses a trade off between the number of pins needed on the voter and the number of cycles required to

complete a read. When the word size is as large as 32-bits, performing the read completely serially will make the design too slow. Performing the read totally in parallel will require such a large number of pins that the design will not be implementable on a chip. Hence, our design uses a mixed mode of data exchange. Every word of data is read serially one byte at a time, starting with the least significant byte. To conserve the number of pins, the voter does not provide a distinct port for every distinct word of data it must read. Rather, it provides only one (byte-wide) input port for every distinct sender, i.e., voter and processor, from which the voter must input.

The voter outputs to the processor an IC vector and a status information vector. The IC vector has three elements in it, one element for every processor not associated with this voter. Note that it is not necessary for the voter to send the private value corresponding to its own processor as the processor already has the value. The voter is designed so that it does not even compute a majority on the value associated with its own processor.

The voter operates in one of five states. In the "idle" state, the voter waits and indicates that it is free by keeping the most significant bit of `status` high. The voter remains idle until it receives a "go ahead" signal on the `vstart` input line, upon which it moves to the load state.

In the "load" state, the voter reads the private values from each of its three successor processors on the input lines `p1`, `p2`, and `p3`. It then moves to the "exchange" state in which it sends the data received on `p1`, `p2`, and `p3` out on the `cross` output line, in that order, while reading data on the input lines `v1`, `v2` and `v3`. It reads the data on `v1`, `v2` and `v3` selectively ignoring the bytes that belong to the private value associated with its own processor.

At the end of the exchange state, the voter will have read a total of nine values that are grouped into three read sets, one for every processor outside the voter's region. In the "compute" state, the voter compares the three values in each of the read sets to produce three majority values. The majority value of a set is computed by taking the majority of the three values in the set. In the "output" state, the voter outputs the results of the IC computation to the processor.

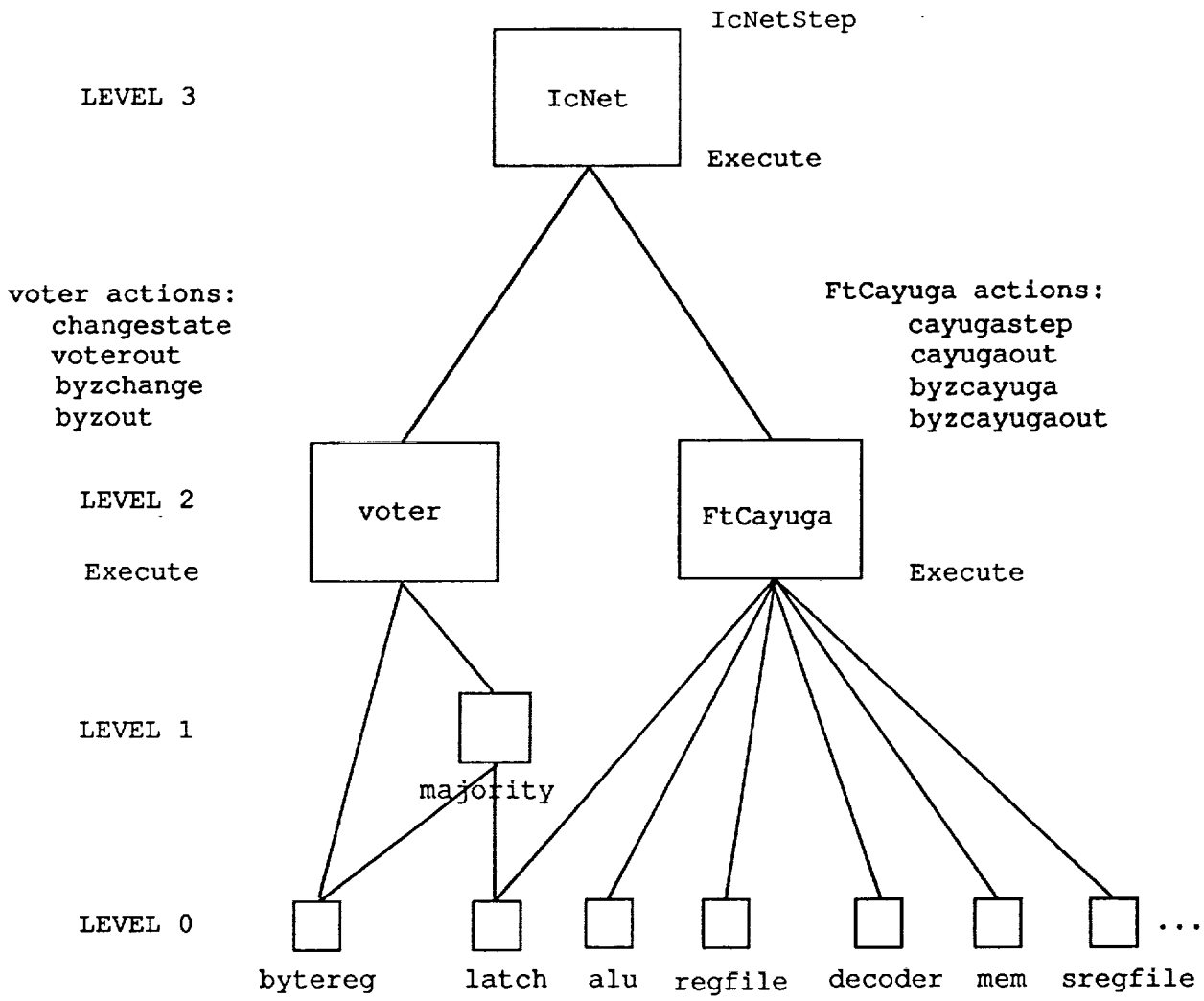


Figure 2: The Design/Specification Hierarchy

5 Overview of Results

Before presenting the complete details, we give an overview of the formalization and the verification of our system.

5.1 The Design and Specification

The hardware system is designed as a multi-level hierarchy, as shown in Figure 2. Every (non-primitive) block in the hierarchy is specified in Caliban at two levels: abstract level and

design level. The abstract specifications, which are written manually, describe the behavior of the blocks by hiding (and, possibly, restructuring) some of the details of the designs. The abstract specifications were constructed primarily to decompose the proofs into smaller units and present the designs more intuitively than the detailed design specifications. The unit of time at which actions are viewed is also abstracted. For example, while the smallest unit of time in the design specification of voter and processor corresponds to one phase on a 4-phase clock, the smallest unit of time in the design specification of `IcNet` corresponds to a clock period.

We described the design at a level of hardware representation that we refer to as a *finite state controller* circuit; this level of detail is comparable to a register transfer level description of hardware. A finite state controller circuit consists of a *data path* and a *controller*. The data path for a block is constructed by connecting one or more instances of blocks at the next level below. The controller is described as a finite state machine that co-ordinates the actions on the data path blocks. The operational behavior of a finite state controller circuit model of hardware is described in section 7. The formal design specification is generated automatically by Spectool from a graphical representation of a finite state controller circuit in a standard fashion. The standard structure of a design specification for a circuit is described in the Appendix A.

We consider a *fault* to be a condition in which a piece of hardware is not operating within its specifications and an *error* to be an incorrect operation due to a flaw in the design of the hardware. When a fault occurs in a block, errors may or may not be produced. Hence, the state and the outputs of the block must be considered to be unknown. In our formalization, we model the possible occurrence of component hardware faults and the unknown nature of the resulting state of the hardware. It is important to note that this modeling is for specification purposes *only* and is not part of the hardware design. Thus, whenever the formal specifications consider the two cases of whether a component is faulty or not, this analysis is not intended to be part of the hardware logic.

The possibility of a region of `IcNet` being faulty is modeled by means of the predicate `faulty` on the indices associated with the fault regions. The term `faulty(ind)` is true if and only if the region with index `ind` is faulty. Fault modeling is used only in the design and abstract specifications of `IcNet` in the design tree because this is the only place where the hardware is replicated and the condition to be verified is an interactive consistency criterion. At other levels there is no replication in the design, and hence fault modeling is not needed

in the specifications.

5.2 Verification

What we mechanically verified is that the system performs the interactive consistency computation correctly provided the components in the system are synchronized properly to execute the ICOP instructions and at most one region is faulty during the computation. More specifically, we verified about the design the following *IC correctness condition.* The condition relates two states of the system that are 12 cycles apart.

```
Assuming
  (1) there is at most one faulty region,
  (2) the nonfaulty processors have all advanced to execute the
      writeback stage of the special ICOP instruction, and
  (3) the nonfaulty voters are all synchronized to be in idle state,
then 12 cycles later
  (1) the nonfaulty voters will again be idle, and
  (2) the IC vectors of the nonfaulty processors will satisfy the
      unanimity and the correctness criteria for interactive consistency
      (The IC vector for a processor is defined to be the value of SREG[0]
      at the beginning of the IC computation followed by the contents
      of SREG[1], SREG[2], SREG[3] at the end of the IC computation.)
```

The behavior of a circuit is defined formally as the set of all possible legal (*infinite*) traces of the state of the circuit and its outputs generated over time (clock cycles) as the input signals change over time. At the design level the generation of a single step of a trace is specified by means of an "Execute" function. At the abstract level, the same is specified by means of a set of "Step" functions. The single cycle behavior of the entire system, i.e., `IcNet`, is defined by the function `IcNetStep`.

The verification is carried out hierarchically. The IC correctness condition is proved as a property of `IcNetStep` (iterated 12 times). The design of every (non-primitive) block is verified separately by proving that its "Execute" function implements its "Step" functions correctly. To verify the design of a block with respect to its abstract specification, an abstraction function (`ABS`) is defined to relate a design state to its corresponding representation in the abstract specification. An advantage of such a hierarchical approach is that the

verification can be done incrementally one block at a time. The implementation correctness has been verified for IcNet and Voter, but not yet for FtCayuga. Thus, our verification guarantees that the system as a whole satisfies the IC correctness condition for any finite state controller circuit design for FtCayuga that meets the given abstract specification for FtCayuga.

We deferred the verification of FtCayuga for the following reasons. The abstract specification that we have developed for FtCayuga is much closer to a design specification than a traditional instruction level description of a processor. Secondly, we plan to enhance certain other parts of FtCayuga hardware that do not deal with interactive consistency to make FtCayuga more versatile. So, we decided to postpone the verification until all the enhancements were completed.

A top level formalization in Clio of the IC correctness condition is given below. More details are given in section 9.

```
MainTheorem := Preconditions 's'
                => Consistent 'icvec s (Iterate #12 IcNetStep s)'

Preconditions 's' := Proper_icnet 's' & Sync 'LDP1' 's' & All_go 's'
```

The predicate Preconditions formalizes the preconditions listed in the informal correctness statement given earlier. Sync and All_go define the preconditions on the voters and processors, respectively. Proper_icnet is a condition on the system state that is separately proved to be true in every cycle. This condition includes certain general properties about our specification model. Hence, it can be used to strengthen the preconditions of the theorem. The requirement that at most one region is faulty does not explicitly appear as a precondition in the MainTheorem because the predicate faulty, which is used inside Sync and All_go to assert a region being faulty, is itself defined so that at most one region can be faulty. The proof proceeds by a case analysis on the five possible "at most one fault" situations.

The predicate Consistent formalizes the IC correctness criteria (section 3) as a condition on the vector of IC vectors of the processors. The vector of IC vectors is constructed by the function icvec from the states of the processors at the beginning and the end of the IC computation.

The correctness condition ensures that, once the processors have advanced to the point of executing the writeback stage of an ICOP instruction, the IC computation gets completed correctly. The condition guarantees that, when an IC computation is underway, neither the occurrence of interrupts nor the execution of any other instructions, including another ICOP, will interfere with the IC computation.

The correctness condition is not sufficient to ensure that the microprocessor system can be programmed to reliably accomplish the interactive consistency task under all circumstances. To see what other conditions must be proved, one must consider the different ways in which the preconditions to the property we have verified may be violated. For example, the clocks driving the replicated resources may not all be synchronized. Since the clock signals are implicit in our model, proving that the clocks are synchronized is outside the scope of the present framework. It is best to separate this task since clock synchronization circuits are designed and implemented separately. Some of the other conditions that are within the scope of the framework that must be proved are the following.

1. The only way in which an IC computation gets initiated must be by means of an ICOP instruction. The destination registers in SREG, where the results of an IC computation will be stored, must be protected from being written into by any other means. These conditions are required to ensure that the results of an IC computation will not get changed before they are properly used.

2. There must be a way of initializing the system to a state that satisfies the precondition to the `MainTheorem`. For this it is necessary to design a special resetting sequence that must be proved to accomplish the objective. The current design does not support such a reset.

3. An IC computation must leave the system in a state in which a new IC computation can be properly initiated. That is, the precondition to the `MainTheorem` must again hold at the end of the computation. We have proved that the `Proper_icnet` condition holds again. The rest of the precondition should follow from some of the lemmas we had to prove in establishing the `MainTheorem`, but must be mechanically checked.

4. The hardware added to the processor to deal with interactive consistency must not affect the correctness of the other instructions.

6 Related Work

In the past few years, we have seen the formal verification of several significant hardware designs—[5], [4], and [8], to cite a few related to microprocessors—described at a level comparable to ours. None of the microprocessor efforts so far have dealt with replicated fault-tolerant designs. The work that comes closest to ours is another NASA sponsored effort [1] in which an interactive consistency design was independently verified using the Boyer-Moore theorem prover. The hardware design verified in [1] is a special-purpose circuit—similar to our voters, but more abstract—that is devoted primarily to interactive consistency. The design in [1] does not deal with the support of interactive consistency as an instruction on a microprocessor and the required interaction between the voting circuit and the replicated microprocessors. In [1] an abstract implementation of the algorithm [6] for the most general case is also mechanically verified.

7 The Hardware Design Model

We refer to our representation model of hardware designs as a finite state controller circuit. A finite state controller circuit consists of a controller part and a data path part. For example, a finite state controller circuit design of the voter is shown in Figure 4.

The data path is a set of interconnected components. Some of the components can be connected to input and output ports for external communication. Every component is assumed to support a set of *actions* that collectively define the behavior of the component. An action, when triggered on a component, may cause a change in the internal state and/or the state of the signals at the outputs of a component. The effect of an action is a function of the current state and current inputs of the component. For example, a `latch` has a single action, `Set`, which has the effect of updating the internal state of the `latch` by the signal that appears at the input wire of the `latch`. A component denoting an arithmetic logic unit (`ALU`) can have an action for every arithmetic/logical operation it supports.

The controller coordinates the actions in the data path. It is described as a finite state machine that schedules a set of actions in each of its states. Thus, actions defined on a component are abstractions of control signals accepted by the component. The inputs to the controller can originate from the data path or from outside the circuit. The set of

actions scheduled is defined as a function of the current controller state and inputs. Every action scheduled in a state is assumed to be triggered simultaneously upon a controller state transition. A controller state transition corresponds to an advancement of discrete time (a cycle) on the implicit global clock. Section 7.1 describes our timing model in more detail.

The behavior of a finite state controller circuit is defined as a *trace* of the *state* of the circuit as its inputs change over time. The state of a circuit may include its outputs and the states of the controller and data path components. The behavior of a finite state controller circuit is uniquely determined given the following:

- The data path and controller structure.
- The controller schedule.
- A specification of the effect of the actions on every component.

It is possible to write a generic formal specification for a finite state controller circuit that derives the behavior of the circuit given the information listed above, which is precisely what Spectool does. In Spectool, the user presents the information listed above graphically, with appropriate textual annotations for specifying the actions.

7.1 The Timing Model

The specification models the design at a *synchronous level* of timing. In a hardware implementation of the design, the actions on the controller and other components will actually be triggered by some synchronizing event, e.g., the *active* edge, of a master clock. The clock, however, remains implicit in our model. We use a discrete notion of time that has a fixed relation with every state transition of the controller. We associate every *cycle* of the clock with the state in which the controller will be for the duration of the cycle. Every clock cycle may be further subdivided into a finite number of phases of equal duration (we use four phases). The actions scheduled by the controller in a state are distributed among the four phases. The lowest granularity of the discrete time used in the design specifications corresponds to the duration of a phase of a clock cycle.

The actions on the components may introduce different amounts of delay measured in integral numbers of phases. If an action, with a delay of δ (≥ 0) units, is scheduled on a component in phase t of a cycle, then the effect of the action is guaranteed to be completed

and the "results," i.e., state and outputs, available for use in phase t+δ. The inputs needed for an action must be ready in the phase in which the action is scheduled. An action may have "zero" delay. Such an action must be implemented by a combinational block whose outputs are assumed to be available in the same phase the action is scheduled. For example, in our design the actions on a multiplexer and the action `scheduler`, i.e., the combinational logic in the controller that determines the control signals, have zero delay.

Between the time t+1 and t+δ-1 (for $\delta > 1$), when an action is still underway on a component, we assign its state and outputs an "undefined" value, although in reality some arbitrary signal value might actually be present. The constant `bottom` of Caliban is used to represent the "undefined" value. We define the effect of performing an action on a component in a `bottom` state or with a `bottom` input to be always `bottom`. This convention allows us to perform a form of timing analysis on our design. We can ensure that the following conditions will not be violated by the design by verifying that no part of the design is `bottom` after every action in a given schedule of actions is completed.

1. No action will be started on a component when another action is still underway on the component.

2. When an action, a, is underway on a component no other action that is dependent on the results of a will be started before a is completed.

A detailed discussion of some of the timing constraints that our abstract timing model imposes on a lower level hardware representation of our designs is given in Appendix B. The constraints can be summarized as follows:

- The real delay introduced in practice by the combinational logic implementing our zero delay actions must be small enough with respect to the actual clock period chosen. If distinct clocks are driving different parts of the circuit, then the skew between the clocks must be similarly bounded. A more precise characterization of the constraint in terms of worst case "setup" time for lower level devices is given in Appendix B.

- To overcome a timing problem that can arise when there is a significant skew between the clocks driving two distinct blocks that exchange data, we employ the following conservative convention in our designs: *The sender must hold its signal for one additional phase following the phase in which the receiver is supposed to use the signal.*

8 The Specification

The entire system is constructed and described as three finite state controller circuits that implement each of the three main blocks—`IcNet`, `FtCayuga`, `Voter`—in the design hierarchy shown in Figure 2 (section 5). `IcNet` is constructed from four instances each of a `Voter` and an `FtCayuga`. A voter is built from a `majority` block (3 instances) and a `bytereg` block (3X3 = 9 instances). A `FtCayuga` is built from blocks, such as `alu`, `regfile`, `decoder`, etc., that are normally found in a block level design of a microprocessor.

Note that the `Voter` and `FtCayuga` blocks have a dual status. That is, at one level they are designed as nontrivial finite state controller circuits and at another level they are used as data path components in the design of another block. Hence, two different specifications—design and abstract—that appropriately describe their behavior at the two levels are developed for each. The `IcNet` specification is developed by composing the abstract specifications of the voter and processor. The abstract specifications hide and reorganize some of the design details that appear in the design specifications. Having two specifications for the blocks decomposes the system specification into smaller and more readable units. It also simplifies the overall correctness proof by decomposing it into separate and smaller units. For similar reasons, an abstract specification was also constructed for `IcNet` although the IC correctness condition could have been verified directly as a property of the design specification of `IcNet`. The remaining blocks shown in Figure 2 are treated as primitive blocks; therefore, they are given only an abstract specification.

The abstract specification of a block defines a set of actions that together capture the intended external behavior of the block. For example, the `alu` block supports actions that perform the various arithmetic and logic operations, such as add, subtract, etc. The `regfile` block supports actions to read from and write into the registers in the register file. The normal behavior of `Voter` and `FtCayuga` are specified by "step" and "out" actions. The "step" action defines the effect on the state of the block for a single clock cycle. The "out" action defines the outputs produced by the block in a clock cycle. The blocks also support an additional set of pseudo actions whose names are prefixed with the string "byz." The "byz" actions, which are left unspecified, are used to denote the effect on the state and the outputs of the block when the block is faulty. The design specification for a block defines the design behavior by means of an "`Execute`" function. The "`Execute`" function for a block is defined by composing the actions of the blocks used in the design.

The granularity of time gets more abstract, i.e., coarser, as we move up the design hierarchy. The smallest unit of time used in the design specifications of the voter and processor corresponds to the duration of a phase of a 4-phase clock implicit in the designs. The "`Execute`" functions for voter and processor define the cumulative behavior of the designs for a single clock period. The unit of time used in the specification of actions on the voter and processor, as well as in the design of `IcNet`, is a single clock period.

Note that the voter and processor circuits, in reality, consume and produce signals (and change states) in every phase of a clock. Hence, a consequence of abstracting time to a clock period in a specification is that the value of a signal at any time instant must be represented in the specification as a 4-tuple, where each field of the tuple indicates the value for the signal in a distinct phase of the clock. This introduces a certain degree of artificiality to the way in which the input signals to an abstract voter and processor are interpreted. The "current input" of a component is a tuple of the values the signal at the input has in the four phases of the current cycle. A "step" action defines the next state as a function of the current state and the current input viewed as a tuple. For this reason, while composing the abstract specifications of voters and processors, the "out" actions must all be done first on the components before the "step" actions.

It must be noted that it is not possible to meaningfully apply such a temporal abstraction in all situations. This abstraction is possible for the voter and processor because their outputs produced in each of the four phases of a clock cycle are determined as a function of only the state at the beginning of a cycle. That is, none of the outputs is dependent on the intermediate states of the block within a cycle. Similarly, the state at the beginning of the next cycle is not dependent on the intermediate states.

We decided to employ the above mechanism, although it appears a bit tricky, because the alternative method involves composing the voters and processors at the phase level. A phase level composition would substantially increase the number of states to be considered in a specification, with an increase in the complexity of verification as well.

The next three sections describe the design and abstract specifications of the three major blocks of the system. In the abstract specification, the behavior of the block is defined using an abstract representation for the state. The design specification is expressed in terms of the data path and finite state controller used in an actual design. For each of the blocks, we first give an informal description of the finite state controller circuit implementing the block.

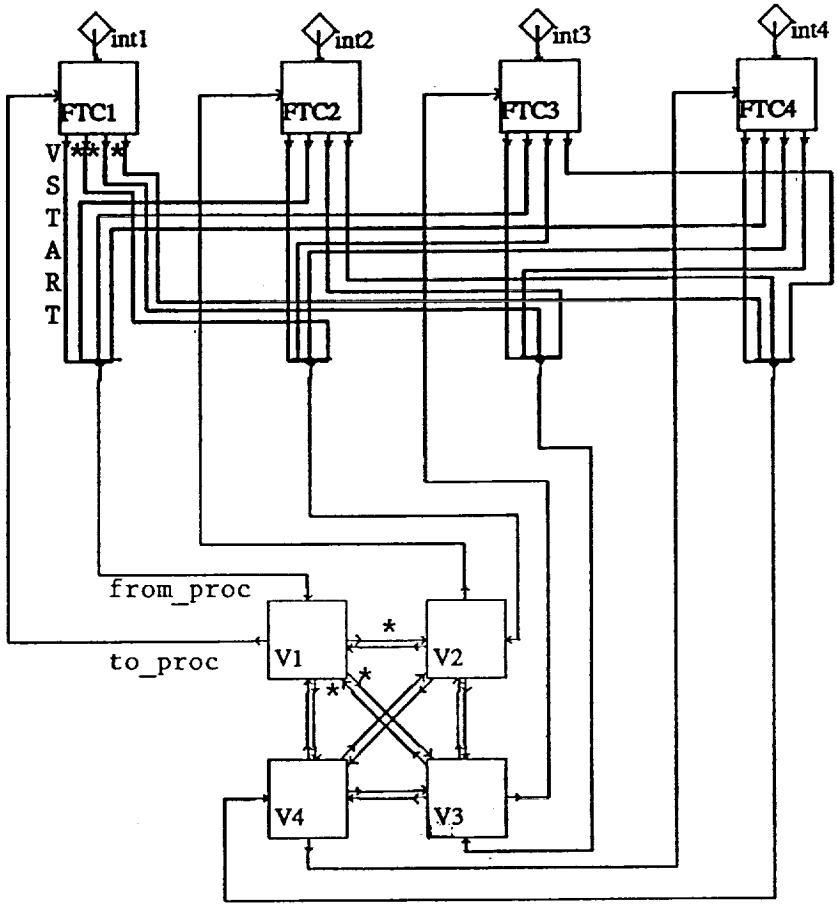
Following that informal description, we give a top level part of the abstract specification for the block along with a brief explanation of the specification. We have not provided a description of the design specifications as they are generated by Spectool in a standard fashion from the circuit diagrams. Appendix A gives a description of the general structure of the design specification generated by Spectool. The full text of the specifications is given in [3].

8.1 Specification of IcNet

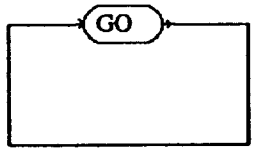
The data path for `IcNet` is shown in Figure 3.

The voter and processor components used at this level of design are abstracted versions of the actual voter and processor designs. The main abstractions employed are described below. Some of the individual inputs (and outputs) that appear in the voter and processor designs are combined into a single compound input (and output). All the outputs of a voter intended for a processor are combined into a single output (`to_proc`). The corresponding inputs of a processor are also combined. (Hence only a single wire is shown carrying the signals from the voters to the processors in Figure 3.) All the inputs that a voter expects to get from a processor are combined into a single input (`from_proc`). The special T-shaped component in Figure 3 is used to combine a sequence of signals into a single compound signal.

Note that the connections that replicate, i.e., fan out, the output signal denoting the private value of a processor to voters in other regions are encapsulated within the abstract processor component. Similarly, the connections that fan out the output from a voter intended for voters in other regions are encapsulated within the abstract voter component. Hence every voter and processor has three additional outputs that normally carry the same value. The replicated outputs from a voter and a processor are marked with an asterisk in Figure 3. (In the figure, the asterisks are shown only for one voter and one processor, since the others follow a similar pattern.) This encapsulation makes it possible to capture in our specification the possibility of a failure in a wire connecting two regions as a failure of the component that is producing a signal on the wire. To specify this failure, we assign either a normal or an unknown fault signal value for each of the additional ports.



Datapath



Controller State Machine

Figure 3: IcNet Circuit Diagram

8.1.1 Abstract Specification

The abstract specification defines the behavior of the system by means of the function IcNetStep. IcNetStep defines the state of IcNet in the next cycle as a function of the state in the current cycle after accounting for the possibility of faults in the system. The state is represented as a tuple that includes the states of the voters and processors, which are structured as aggregates. The state of the aggregate of voters (or processors) is represented as a function from region indexes (type INDEX) to the current state of the individual voter (or processor) in the aggregate. For convenience, the only input to IcNet, a set of four interrupt signals meant for the processors, is also included as part of the state of IcNet. The input component in the state is represented as a list that can be arbitrarily long. The list contains the values for the input signal over time starting from the current cycle. Defining IcNetStep as a function over a list of input values, instead of only on the current input, is an advantage because it is easier to express multi-cycle behavior of the system by iterating IcNetStep over the list.

```
||********************** Step function for IcNet ****************************
|| The abstract state change (behavior) of the voter-net.
|| What happens at each index depends on whether that index is faulty.
IcNetStep <<ftc,vtr, int:rest>> =
     <<newftc,newvtr ,rest>>
     where newftc index = fault_ftc_step index ftc (ftcinput index )
           newvtr index = fault_vtr_step index vtr (vtrinput index )
           ftcinput index = make_ftc_in (select_int index int)
                                        (fault_to_proc index ftc vtr)
           vtrinput index = Voterinput index ftc vtr


fault_ftc_step index s in =
    FtCayugaStep (s index) in , ~(faulty index)
    byzCayugaStep (s index) in

fault_vtr_step index s = voterstep (s index) , ~(faulty index)
                         byzstep (s index)

fault_to_proc index ftc vtr =
    to_proc (vtr index) , ~(faulty index)
```

```
    byz_to_proc (vtr index)

Voterinput index ftc vtr =
    <<fault_from_proc index ftc,
      fault_cross THREE (succ3 index) vtr,
      fault_cross TWO (succ2 index) vtr,
      fault_cross ONE (succ index) vtr>>

fault_from_proc index ftc =
    <<fault_vstart index ftc,
      fault_prvt (whichprvt index (succ index)) (succ index) ftc ,
      fault_prvt (whichprvt index (succ2 index)) (succ2 index) ftc,
      fault_prvt (whichprvt index (succ3 index)) (succ3 index) ftc>>
```

The value that `IcNetStep` returns is defined in terms of the following set of functions: `fault_ftc_step`, `fault_vtr_step`, `fault_to_proc`, and `Voterinput`. The first two of these functions define the next state for the processor and the voter aggregates, respectively, as a function of their current state and inputs. The last two define the current inputs to the processor and voter aggregates, respectively, being produced from the other aggregate. In other words, they define, respectively, the aggregate of outputs of the processors and voters. (Note that `make_ftc_in` constructs the complete input to a processor by combining the external interrupt input and the inputs being produced by the voters.)

The functions mentioned above take into account the possible occurrence of faults. These functions are defined in terms of the functions that specify the normal behavior and faulty behavior of the voter and processor. `FtCayugaStep` and `from_proc` define the normal behavior, i.e., next state and outputs, for `FtCayuga`. The functions `voterstep`, `cross`, and `to_proc` define the normal behavior of the voters. The functions that are prefixed with the string "byz" define the faulty behavior of the voter and processor. There is one such function to denote the state and every output of the voter and processor. The functions denoting the normal behavior are defined in the abstract specification for the voter and processor. The "byz" functions are left unspecified, which has the effect of making no assumptions about the state and outputs of a faulty component except that the state and outputs have some unknown values.

The possibility of a region of IcNet being faulty is modeled by means of the predicate faulty on the type INDEX— the term faulty(ind) is true if and only if the region with index ind is faulty. That is, one or more components in the region ind are faulty. The predicate faulty is defined so that at most one region can be faulty. We define faulty in terms of an unspecified constant, the_fault (of type FAULT), which may take one of five values denoting the five possible "at most one fault" situations: one or none of the four regions being faulty. The specification of the predicate faulty is given below.

```
||********************** Fault modeling ********************************
|| The possible faults are listed in the following enumerated type:

FAULT ::= Region !INDEX | NO_FAULT

|| To model the fact that we are assuming at most one fault, we can suppose
|| that there is a constant, "the_fault" of type FAULT, and then define the
|| predicate "faulty" in terms of that constant.

the_fault :: FAULT
AXIOM '!the_fault'='True'

faulty index = (the_fault = (Region index))
```

8.1.2 Design Specification

We generated the design specification using Spectool by constructing the data path and controller shown in Figure 3. The controller used in the design of IcNet is purely symbolic in the sense that it does not translate into any real hardware. The controller is constructed for two reasons. First, to generate a design specification using Spectool, it is necessary to have a controller to activate the data path. Second, having a controller makes it possible to conveniently specify the behavior in the presence of faults.

The controller consists of just a single state GO. If any one of the regions is faulty it schedules the respective "byz" actions on every component in the region. Otherwise, the controller schedules the normal actions on the components. The design specification uses the same formal machinery for fault modeling as the abstract specification uses. That is, the

controller actions are conditioned on the predicate `faulty` shown earlier.

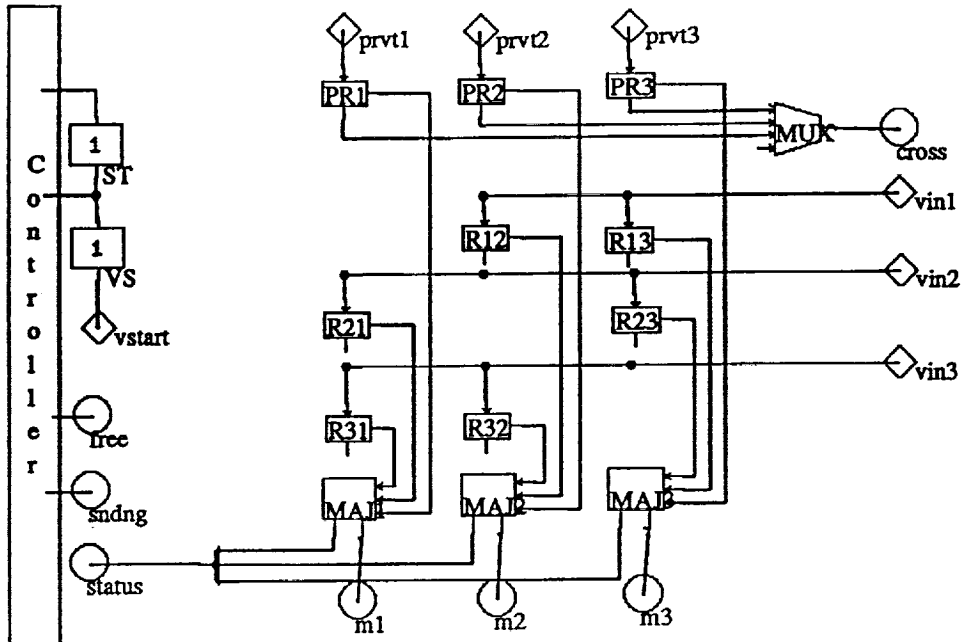
While scheduling the actions, the controller must schedule the actions that produce the outputs on all the components before scheduling the actions that advance the state of the components. One way to accomplish this scheduling in Spectool is to assign the "Out" actions zero delay and the "Step" actions unit delay.

8.2 Specification of the Voter Circuit

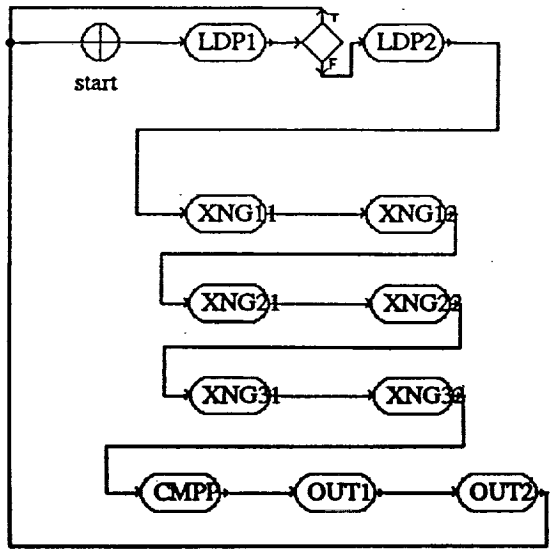
Figure 4 shows the circuit diagram of the voter. A voter has the following inputs and outputs:

- `vstart` is a 1-bit input through which the voter gets the signal to start the voting process.
- `prvt1`, `prvt2`, and `prvt3` are byte-wide inputs through which the voter reads the private values from the out-of-region processors.
- `vin1`, `vin2`, and `vin3` are byte-wide inputs through which the voter reads the second round data from the other voters.
- `free` is a 1-bit output that is true as long as the voter is free to start another voting process.
- `cross` is a byte-wide output through which the voter outputs its first round data to the other voters.
- `sndng` is a 1-bit output that is true when the voter is ready to send the three consensus values through the byte-wide outputs `m1`, `m2`, and `m3`.
- `status` is a 3-bit output that indicates if the voter was able to determine a majority value for each of the three outputs `m1`, `m2`, and `m3`.

The data path of a voter is constructed from the following component classes: `bytereg`, `majority`, `bitlatch` and `mux`. A `bytereg` stores a single word (four bytes) of data and is like an ordinary register. But, it has actions `seti` (unit delay) and `readi` (zero delay) to set (from the input at the top) and read out (from the output at the bottom) each of the four bytes in the word separately. The entire stored word is available at any time at the side output. The voter has a total of nine instances of `bytereg`: `PR1`, `PR2`, and `PR3` (*private registers*) to store the private values read in directly from the processors; and the rest (*cross registers*) to store the values (two from each of the voters in the other regions) read in the second round. (The voter uses `Rij` to store the private value that the processor $succ^j$ sent to voter $succ^i$ in Round 1. This voter reads this value in Round 2 at input `vini`.)



Datapath



Controller State Machine

Figure 4: Voter Circuit Diagram

A `majority` block supports the following actions: `comp` (unit delay) computes the "majority" of the three inputs (coming in at the side of the icon representing the block) and sets the internal state to the majority value. The majority, when it exists, is the input that is identical to at least one other input. The 1-bit status output (bottom left of the icon) is set to a '1' iff there exists a majority. If a majority does not exist for the inputs, the only assumption made about the computed majority value is that the value must be the same regardless of the order in which the inputs are presented. The `majority` block has four (zero delay) actions, `selecti`, to select each of the four bytes of the majority data stored in the block.

The controller first sets the private registers to the values read from `prvti` inputs. Then, it sends out the values in the private registers while setting the cross registers to the inputs read at `vini` input ports. Then the controller computes majority and sends them out. The control logic is involved because a transfer of data to and from a register must take place one byte at a time. We use two phases per transfer, although the `seti` action has only a unit delay. This strategy follows the safe design convention (appendix B.2) of holding the output signal for one phase beyond the phase in which the signal is used. The data associated with the voter's own processor, coming in on each of the `vini` inputs must be ignored. This data arrives at different times at different `vini` ports. Finally, the voter must perform a handshake with its processor at the start and finish of the voting process. The actions scheduled in the various states of the controller are summarized below. For ease of presentation, we group the controller states into four *stages.* The actions performed in the states within a stage are similar and are logically related.

1. *Load stage* (`LDP1`, `LDP2`): In `LDP1`, with the bit latch `VS` in '0' state, the controller idles until `vstart` becomes a '1'. (That is, the controller takes no action, except, in phase 2, to set `VS` to the value of `vstart`. Thus, the controller reads `vstart` only in phase 2 of a cycle.) The controller holds the output `free` '1' as long as the voter idles, keeping `free` '0', otherwise. With `VS` set to '1', in phases 0 and 2, the controller initiates actions to set the two least significant bytes of the private registers. In `LDP2`, in phases 0 and 2, the controller initiates actions to set the two most significant bytes of the private registers.
2. *Exchange stage* (`XNGi1`, `XNGi2`, $1 \leq i \leq 3$): In `XNGi1`, the controller initiates the following set of actions: select (in phases 0 and 2) the two bytes in the least significant half of `PRi` and output them (via `MUX`) over `cross`; set (in phases 1 and 3) the two bytes in the least significant half of the two appropriate cross registers. The controller

initiates a similar set of actions in XNGi2 except that the most significant half of the registers are involved.

3. *Compute stage* (CMPP): In the CMPP state, in phase 0, the controller initiates the comp action on each of the majority blocks.
4. *Output stage* (OUT1, OUT2): In OUT1 and OUT2, in phases 0 and 2, the controller initiates actions to read out (onto m1, m2 and m3) the least and the most significant bytes, respectively, of the result of the previous comp action on the majority components. The output sndng, which the controller holds at '0' in all other states, is made '1' in OUT1 to indicate that the voter is beginning to send the majority values. At the end of OUT2, VS is reset so that the voter can go back to being idle.

A part of the abstract specification of the voter is given below.

```
VoterStep <<cstate,array,maj,go>>  from_proc  from_voters =
    <<nextstate cstate <<go,go>>, newarray, newmaj, newgo>>
        where
            newarray {(cstate = LDP1) & go} = update_row1 LO array from_proc
            newarray {cstate = LDP2} = update_row1 HI array from_proc
            newarray {cstate = XNG11} = update_diagonal 0 LO array from_voters
            newarray {cstate = XNG12} = update_diagonal 0 HI array from_voters
            newarray {cstate = XNG21} = update_diagonal 1 LO array from_voters
            newarray {cstate = XNG22} = update_diagonal 1 HI array from_voters
            newarray {cstate = XNG31} = update_diagonal 2 LO array from_voters
            newarray {cstate = XNG32} = update_diagonal 2 HI array from_voters
            newarray  = array  || otherwise it's unchanged

            newmaj = (cstate = CMPP)->(maj_vector array) ; maj

            newgo = beginof (select (#2) from_proc) , cstate = OUT2
                    beginof (select (#2) from_proc) , (cstate = LDP1) & (~go)
                    go
```

In the abstract specification, the state of the voter is abstracted into a tuple with the states of the various components in the design organized as follows:

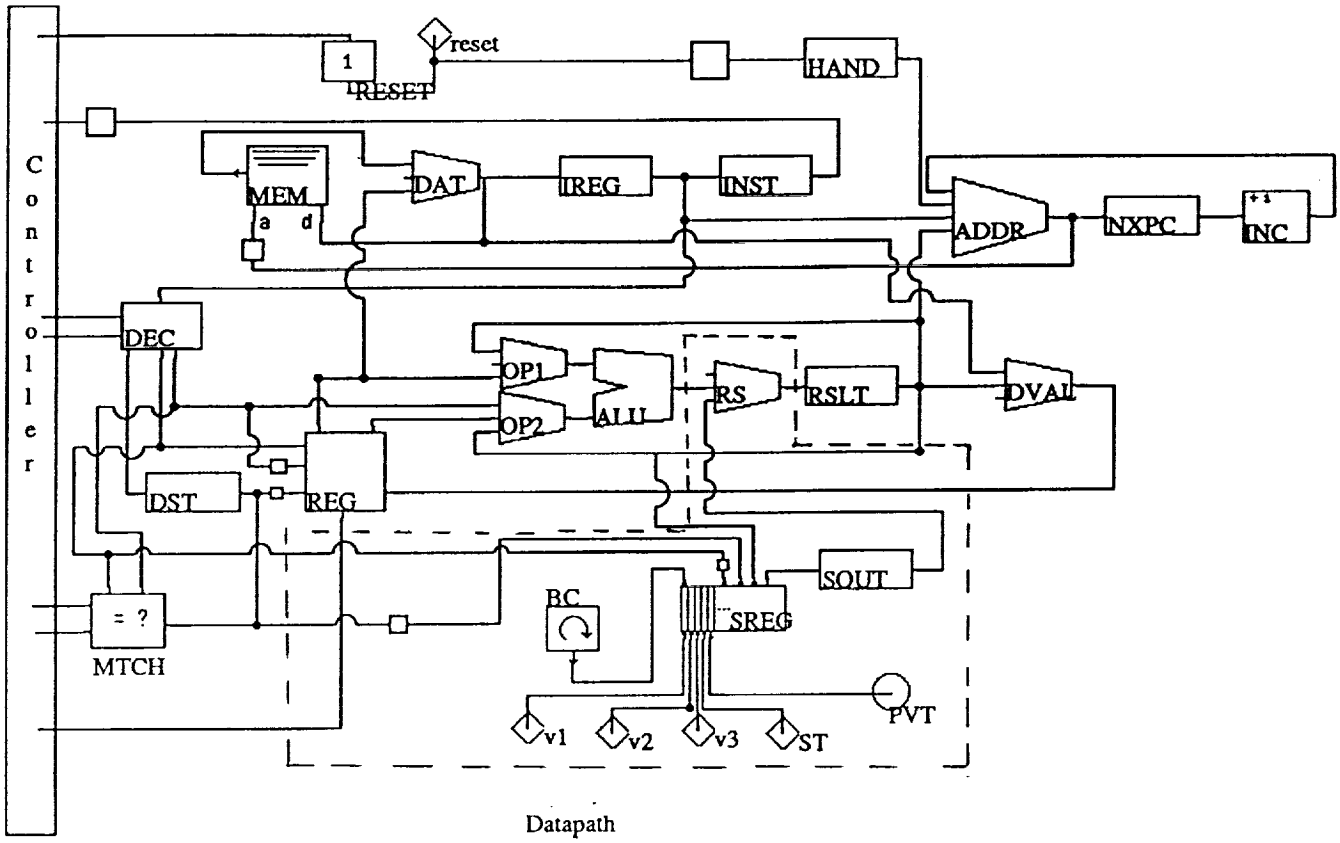
- `cstate` represents the controller state.
- `array` represents the combined state of the array of the private and cross registers in the design.
- `maj` represents the combined states of the `majority` blocks in the design.
- `go` represents the state of the `VS` latch.

`VoterStep` defines the state of the voter in the next clock cycle as a function of the current state and the two abstracted (compound) inputs to the voter, one from the processor and the other from the voters. The outputs of the voter are specified by defining a separate function for every output. We deliberately did not abstract the controller state because the single cycle behavior of a voter is better understood this way. The new values for the components in the abstract state are determined by the various helping functions, such as `nextstate`, `update_row1`, `update_diagonal`, etc., the definitions of which appear in the complete specification. In the design specification, the state transformation is not specified so directly. The design specification defines the behavior by specifying the actions that occur on the data path components in the four phases of a clock cycle.

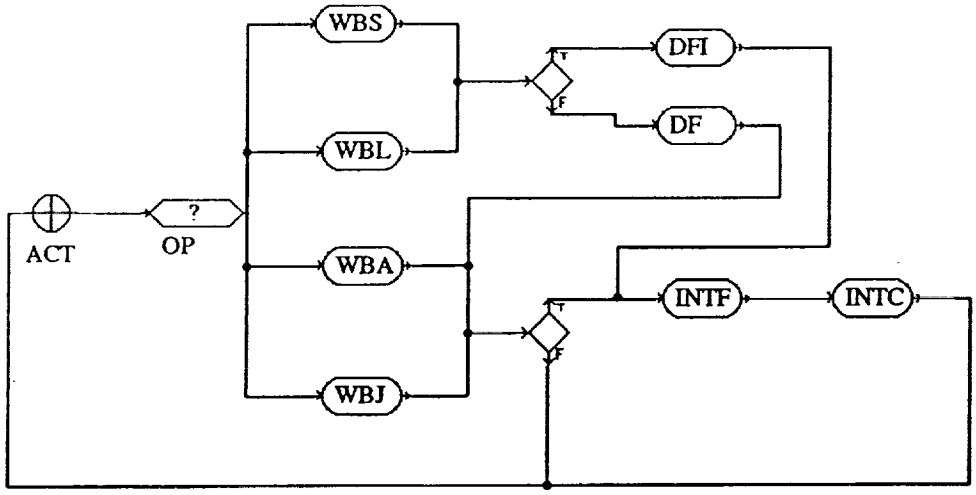
8.3 Specification of the FtCayuga Circuit

Figure 5 shows the circuit diagram of the `FtCayuga` design. The part enclosed within the dotted lines denotes the extra circuitry that we needed to add to obtain the data path of `FtCayuga` from that of MiniCayuga. The state structure and transition relation of the finite state machine implemented by the controller remained the same as in MiniCayuga. We added several new actions to the schedule in every controller state. None of the actions that implemented the old instructions of MiniCayuga inherited by `FtCayuga` had to be altered.

Some of the major blocks in the circuit are the following: `REG` is the register file, which consists of all the programmable registers of the processor; `ALU` is the arithmetic and logic unit; and `DEC` is the instruction decoder. `MEM`, the main memory with which the processor interacts, is not part of the processor, although `MEM` is shown as part of the circuit diagram.



Datapath



Controller State Machine

Figure 5: FtCayuga Circuit Diagram

Some of the new blocks that were added to support the new instructions are the following: SREG is a special register file, the first five registers of which are used for the interactive consistency task; the register indexed by PVT holds the private value used for interactive consistency; the registers with indexes VT1, VT2, and VT3 hold interactive consistency vector components corresponding to the successor processors, in order; and the register STATUS holds the status output from the voter. Since the processor outputs its private value one byte at a time, we need to keep a record of the byte number that is currently being output. BC is a modulo byte number counter used for this purpose.

The instructions of FtCayuga are grouped into four classes: *arithmetic*, *load*, *store*, and *jump*. Instructions belonging to the load/store class are the only ones that move data between memory and registers; the rest are all register to register. The new instructions all belong to the arithmetic class.

FtCayuga uses a three-stage instruction pipeline to execute its instructions. The actions required to execute an instruction are partitioned into three stages: *fetch*, *compute* and *writeback*. In any given cycle, the processor can simultaneously execute the writeback stage of the *current* instruction, the compute stage of the *next* instruction and the fetch stage of the *next-to-next* instruction. Thus, the processor usually completes one instruction every cycle although each instruction takes three cycles.

Normally, the execution of an instruction in the arithmetic class involves the following activities. In the fetch stage, the processor fetches the instruction from MEM and stores it in IREG. In the compute stage, the processor first decodes the instruction, then fetches the source operands from REG, performs the required ALU operation (add, subtract, etc.), and stores the result in RSLT. In the writeback stage, the contents of RSLT is moved to its destination in REG.

The controller is in one of the states labeled with a "WB" prefix when the pipeline is *active*, i.e., when the processor is executing (different stages of) three distinct instructions. The exact state in which the controller will be while the pipeline is active depends on the class of the current instruction that is currently undergoing writeback. The writeback stage of an arithmetic instruction will be executed in WBA. When the writeback is that of a load or store instruction (WBS, WBL), the controller delays fetching the next-to-next instruction to prevent a memory access clash. Hence the controller takes an extra cycle (DF) to execute load and store instructions. The controller enters the remaining states only in case of an

interrupt.

This normal course of activities is changed by the hardware and the new arithmetic class instructions added to provide support for interactive consistency as follows. For all the instructions the fetch stage proceeds as before. For SADD, the compute stage also proceeds as before. The destination of writeback is, however, SREG and not REG. For MOVE, the compute stage consists of moving data into RSLT from SREG instead of REG. The writeback proceeds as before.

For the ICOP instruction, the compute stage consists of actions that initiate interactive consistency operation after setting up things necessary for the operation to proceed properly. The set up actions consist of resetting BC and clearing the STATUS register in SREG appropriately. Interactive consistency is initiated by sending a "go" signal to the voter on vstart output. All of these actions are performed only if the most significant bit of the input ST is true, which indicates that the voter is free to vote. Otherwise, no action is performed in the compute stage. The writeback stage of ICOP has no actions.

The actions required to send the contents of the private register and to input the results sent by the voter upon completion of interactive consistency are initiated automatically in every cycle, if necessary. The actions that output the private value consist of incrementing BC and reading out the byte pointed by BC from the private register in SREG. The actions that update the appropriate registers in SREG with the interactive consistency vectors start when the voter is ready. (The voter's readiness is indicated by the two most significant bits of ST reading a '01'.) The STATUS register of SREG is updated when the interactive consistency cycle is completed, i.e., when the voter becomes free again (most significant bit of ST a '1').

A part of the abstract specification of FtCayuga is given below. The state of the processor in the abstract specification is represented as a tuple that includes the (abstract versions of) states of most of the components used in the design, including the controller. Thus, the specification is not as abstract as an instruction level description of a processor, but is more abstract than the design specification generated by Spectool.

FtCayugaStep defines the state of the processor in the next clock cycle as a function of the current state and the interrupt input. The outputs of the processor are specified by defining a similar function for every output. The new values for the various components of the abstract state are specified directly in the various helping functions, such as newcontr, newreg, newsreg, etc.

```
type FTCAYUGASTATE = <<CONTROLSTATE, data, data, data, data,
                       addr ->data, data, regaddr ->data, data,
                       sregaddr->data,byte_num>>
|| The fields in the abstract state denote the states of the following
|| design components:
||                    <<controller, IREG, DST, RSLT,HAND,
||                      MEM, NXPC, REG, INST,
||                      SREG, BC>>

FtCayugaStep s in =
    <<newcontr (controf s) (interruptof in) (iregof s),
      newireg (controf s) (interruptof in) s,
      newdst (controf s) (dst_of s)(iregof s),
      newrslt (controf s) s,
      newhand (controf s) (interruptof in) (handof s),
      newmem (controf s) s,
      newnxpc (controf s) (interruptof in) s,
      newreg (controf s) s,
      newinst (controf s) (inst_of s)(iregof s),
      newsreg (controf s) in s,
      newbc s>>
```

9 The Correctness Theorems

The verification is carried out hierarchically. The main correctness theorems proved are the following:

1. A `MainTheorem` that formalizes the IC correctness condition as a property of `IcNetStep` (iterated 12 times). The IC correctness condition (section 5.2), as the reader may recall, asserts that the system will correctly achieve interactive consistency, provided the processors and the voters are all synchronized properly to execute an `ICOP` instruction.

2. A family of *bridge theorems*, one set for every nonprimitive block in the specification hierarchy (Figure 2). The bridge theorems for a block ensures that the design specification "correctly implements" the behavior specified in the abstract specification.

We also proved a set of *Properness* conditions, one for every abstract and design specification of the three major blocks in the design. The Properness condition is so called partly because it is defined by a predicate that begins with the string "Proper." The properness condition for a block is a condition on the state of the block that is invariant over time. That is, it is expected to be true in every clock cycle. Since the properness conditions for the blocks are separately established as invariants, they are used in the proof of the `MainTheorem` and bridge theorems by including the appropriate properness predicate as a precondition in the theorem.

One may include any condition that is invariant and potentially useful inside a properness predicate. One condition that is always included is the assertion that the state of a block is "well-defined." That is, no part of the data structure that represents a state (and a signal) may take the value `bottom`, which denotes the "undefined" value in Caliban. Our specification model is formulated so that the only situation in which some part of the state or signal becomes `bottom` is when there is a timing error (see section 7.1) in the controller designed for the block. Hence the theorem that establishes a properness predicate as an invariant for the block is called a "Timing-ok-lemma" for the block.

9.1 The Main Correctness Theorem

The top part of the formal definition of the `MainTheorem` is given below.

```
MainTheorem := Preconditions 's' => ResultConsistent 's'

Preconditions 's' := Proper_icnet 's' & Sync 'LDP1' 's' & All_go 's'

Sync 'cs' '<<ftc,vtr,inlist>>' :=
    ('faulty ONE' = 'False' => 'control (vtr ONE)'='cs')
 & ('faulty TWO' = 'False' =>  'control (vtr TWO)'='cs')
 & ('faulty THREE' = 'False' => 'control (vtr THREE)'='cs')
 & ('faulty FOUR' = 'False' =>'control (vtr FOUR)'='cs')

All_go 's' :=
    ('faulty ONE'='False' =>
        ('go_of (vtr s ONE)'='False'
         & 'go_signal s ONE'='GO' & 'bytecount (ftc s ONE)'='BYTE1'))
```

```
& ('faulty TWO'='False' =>
     ('go_of (vtr s TWO)'='False'
      & 'go_signal s TWO'='GO' & 'bytecount (ftc s TWO)'='BYTE1'))
& ('faulty THREE'='False' =>
     ('go_of (vtr s THREE)'='False'
      & 'go_signal s THREE'='GO' & 'bytecount (ftc s THREE)'='BYTE1'))
& ('faulty FOUR'='False' =>
     ('go_of (vtr s FOUR)'='False'
      & 'go_signal s FOUR'='GO' & 'bytecount (ftc s FOUR)'='BYTE1'))
```

The antecedent (`Preconditions`) of the theorem is a conjunction of three conditions:

1. `All_go 's'` formalizes the condition that all the nonfaulty processors must be properly synchronized to initiate interactive consistency. This predicate is stated as a conjunction of the following conditions: (1) `go_signal` asserts that a processor must be sending out a `GO` signal to the voter, which happens when the processor is executing the write-back stage of an `ICOP` instruction, and (2) the predicate `bytecount` asserts that the `BC` register of a processor must be reset to the value `BYTE1` to ensure that the data output starts from the least significant byte.
2. `Sync 'LDP1'` formalizes the condition that every nonfaulty voter must be ready to start a new voting process. This condition is stated by requiring that the controller part of the abstract voter state must be in `LDP1` state and the `go` part must be `False`. The latter is actually asserted by the predicate `go_of` inside `All_go`.
3. `Proper_icnet 's'` is the Properness predicate defined for the `IcNet` abstract state.

A top level specification of `ResultConsistent`, the consequent of `MainTheorem`, is shown below `ResultConsistent` defines the condition that the result of an IC computation, namely the interactive consistency vectors of the nonfaulty processors, must satisfy. This condition was informally described as the unanimous and correctness criteria for the interactive consistency algorithm in section 3.

```
ResultConsistent 's' := Consistent 'icvec s (Iterate #12 IcNetStep s)'

Consistent 'array' :=
   'faulty ONE'='False'=> IndexConsistent 'array' 'ONE'
 & 'faulty TWO'='False'=> IndexConsistent 'array' 'TWO'
 & 'faulty THREE'='False'=> IndexConsistent 'array' 'THREE'
 & 'faulty FOUR'='False'=> IndexConsistent 'array' 'FOUR'

IndexConsistent 'array' 'index' :=
    ('faulty (succ index)'='False'=>
       '(array index).succ'='array (succ index)')
  & ('faulty (succ2 index)'='False'=>
       '(array index).succ2'='array (succ2 index)')
  & ('faulty (succ3 index)'='False'=>
       '(array index).succ3'='array (succ3 index)')

|| Note that the expression "icvec init final index" is actually
|| a function on the type INDEX, and represents the vector of four values.
icvec init final index ONE = actualPVT init index
icvec init final index TWO = get_special_register VT1 final index
icvec init final index THREE = get_special_register VT2 final index
icvec init final index FOUR = get_special_register VT3 final index
```

The function `icvec` constructs the interactive consistency vectors of all the processors given the system state at the beginning and end of an IC computation. The function constructs a vector of vectors `A`, such that `A[i]` is the interactive consistency vector of the i^{th} processor. The vector `A[i]` is organized so that `A[i][1]` gives the i^{th} processor's own private value, and the rest of it gives the values for the successor processors, in increasing order. Organized in this fashion, `A` satisfies the interactive correctness criteria if for all non-faulty indexes, i and j, the vectors `A[i]` and `A[j]` are identical after one of the vectors is cyclically rotated as many times as the distance between `i` and `j` in the cyclic indexing scheme.

The function `actualPVT` used in the definition of `icvec` defines the actual value on which the system performs the IC computation. Note that, since the processor sends the value to the voter over two cycles, we define one half of this value based on the contents of the special register `PVT` in the cycle in which `ICOP` is executed and the other half based on

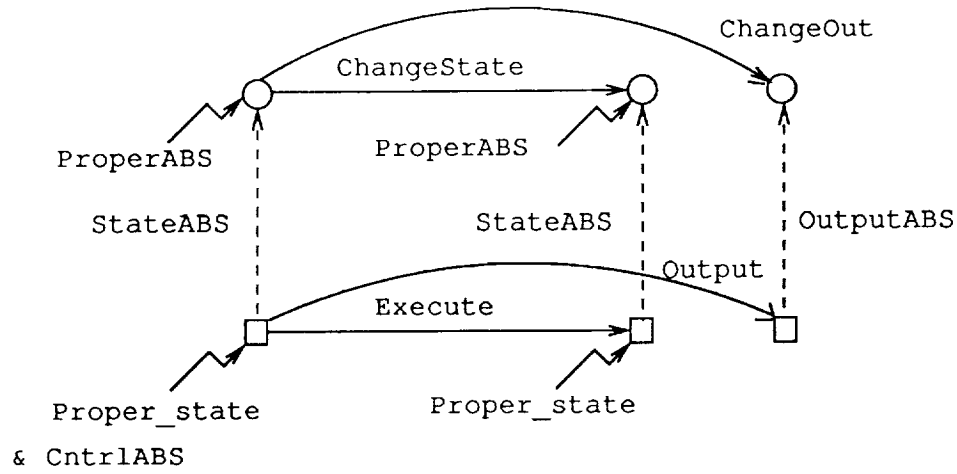


Figure 6: General forms of bridge theorems

the contents of PVT in the following cycle.

9.2 General Theorems for Bridging Abstraction Levels

A set of bridge theorems are proved for a block to show that the design specification correctly implements the behavior described by the abstract specification. Since the interface defined by our design and abstract specifications are standardized, the bridge theorems for every nonprimitive block take a standard form. The general form of these theorems for a block is described below.

Note that the single cycle behavior of a block in the design specification is defined by means of `Execute`, which determines the next state as a function of the current state. The specification also defines an `Output` function that extracts the outputs of the design from the new state. Suppose in an abstract specification that the corresponding functions are `ChangeState` and `ChangeOut` actions. The state and the outputs at the abstract level are represented as abstractions of the state and outputs in the design. To relate the behavior at the two levels it is necessary to define a set of *abstraction functions* that establish the desired correspondence between the representations at the two levels. Suppose `StateABS` is the function that maps a design state to the abstract state it corresponds with, and `OutputABS` does the same for the outputs. Then, the relation that must be established to ensure that a design specification is a correct implementation of an abstract specification is shown in Figure 6. The general formulation of the bridge theorems is given below.

```
StateTheorem := Proper_state 's' & CntrlABS 's' =>
                  'StateABS (Execute s)' = 'ChangeState (StateABS s)'

OutTheorem := Proper_state 's' & CntrlABS 's' =>
                  'OutputABS (Output s)' = 'ChangeOut (StateABS s)'

Timing_ok_lemma := Proper_state 's'  => Proper_state 'Execute s'


ProperstateLemma :=
   Proper_state 's' => ProperABS 'StateABS s'
```

The `StateTheorem` expresses the condition implied by the commutative diagram shown in Figure 6 for the state. The condition asserts that if the current states correspond (according to the abstraction functions) at the two levels, then the new state of the circuit a cycle later as determined by `Execute` must correspond to that required by `ChangeState`. The `OutTheorem` expresses a similar condition for the outputs of a block. Note that, in the `OutTheorem`, the operator used to relate the abstract and design entities is `<=` (meaning "at least as much defined as") rather than `=`. The reason for using `<=` is that, at the abstract level, some of the outputs may be deliberately left unspecified. In this case, at the abstract level, the semantics of Caliban automatically assigns these outputs the constant `bottom`, whereas the design may assign some other value that is not `bottom`. Hence the need for the weaker condition.

In general, a couple of preconditions may be attached to the `StateTheorem` and `OutTheorem`. One of the preconditions is the properness predicate for the state (`Properstate`). At the abstract level, sometimes the behavior may appear partitioned into several distinct actions, although at the design level the behavior is combined into a single function. This normally happens when some of the signals at the design level are abstracted (as control signals) at the abstract level. In that case, only an appropriate restriction of the design behavior will correspond to the behavior described by an abstract action. The predicate `CntrlABS` is used to apply this restriction.

Two other theorems that we needed to prove are classified under bridge theorems. The `Timing_ok_lemma` is the theorem that asserts the invariant property of the Properness predicate. The `ProperstateLemma` relates the Properness predicates defined at the two

levels. It asserts that Proper_state at the design level must be strong enough to imply its counterpart, ProperABS, at the next level up.

For illustration, the bridge theorems associated with the IcNet block are shown below. Note that CntrlABS in this case is vacuously true because the abstract behavior was not partitioned into separate actions.

```
|| Bridge theorems for IcNet block
IcNetStepLemma :=
    'IcNetABS (Execute s)'='IcNetStep (IcNetABS s)' , Proper_state 's'

ProperIcNetLemma :=
    Proper_state 's' => Proper_icnet 'IcNetABS s'

Timing_ok_lemma
   Proper_state 's'  => Proper_state 'Execute s'

Proper_icnet '<<ftc,vtr,inlist>>' := Proper_voterstate 'vtr ONE'
                                     & Proper_voterstate 'vtr TWO'
                                     & Proper_voterstate 'vtr THREE'
                                     & Proper_voterstate 'vtr FOUR'
                                     & Proper_ftcayuga 'ftc ONE'
                                     & Proper_ftcayuga 'ftc TWO'
                                     & Proper_ftcayuga 'ftc THREE'
                                     & Proper_ftcayuga 'ftc FOUR'
                                     & Proper_inlist 'inlist'
```

10 About the Proof of Correctness

Each of the proved correctness theorems has the same general form shown below. Hence the same proof strategy (with minor variations) was used to prove them. (In the following 's' denotes an arbitrary state of a block.)

```
Somepreconditions 's' =>
  'Somesimplefn s' = 'Someothersimplefn (iterate somenumb Execute s)'
```

The proof strategy that is used can be paraphrased as "controlled symbolic execution of specification." The strategy consists of the performing following steps in order:

1. The variable s is first instantiated to a suitable structured symbolic constant, with separate constant symbols to denote the various subcomponents of the design state, so that the two sides of the equation in the consequent can be simplified.

2. The conditions on the symbolic constants implied by the preconditions of the assertion are all added as hypothesis (equations).

3. The two sides of the equations are simplified to their irreducible form. The goal is to simplify the expressions so that as many "non-primitive" function symbols are eliminated from them as possible. The resulting expressions will usually be a fairly large conditional expression involving action invocations on the design components.

4. If the two sides of the equations do not simplify to the same term, then a *case split* is performed on all the conditions in the conditional expressions.

The Spectool generates appropriate annotations to the theorem prover so that each of these steps can happen automatically. The user must initiate the steps in that order. This strategy worked in discovering a proof as long as the term obtained as a result of the simplification step was reasonably small. The theorem prover seemed to hit a combinatorial explosion bottleneck in the simplification step or in the case-splitting step when the terms were large. The following techniques had to be used to control the combinatorial explosion problem:

1. The case-split step had to be done manually by intelligently selecting the expressions to case-split on. An initial case split on the controller states and controller inputs helped considerably.

2. The `MainTheorem`, which is a 12-cycle property, was split into a series of lemmas each of which characterized the intended behavior over fewer cycles. These lemmas were then chained together to obtain a proof for the `MainTheorem`.

10.1 Assumptions Made by the Proof

In proving the correctness theorems, it was necessary to make several assumptions, particularly about some of the primitive functions used in the specification. These assumptions

were introduced using the AXIOM mechanism of Clio before starting the proof.

The need for these assumptions arose mainly because some of the primitive functions were deliberately left unspecified. For example, we wanted to make as little commitment as possible about the actual bitvector representation of the various signals in the design. For example, we left the word size, i.e., the maximum size of the values of type data, deliberately unspecified since none of the logic at our level of design is dependent on this information. The only assumption made is that data is a word with four bytes, each of which can be manipulated separately; this assumption is reflected in the definition of the type data shown below. In such a situation it is not always possible to provide a complete Caliban definition for the functions that manipulate the signals. Specifying these functions as a set of AXIOMs allows one to make minimal assumptions about these functions. The AXIOMs express "constraints" on the functions. Every such assumption made must be proved when our design is refined by choosing a specific representation for data.

As an illustration, a few of the assumptions made are shown below. The entire set of assumptions is given in [3]. The assumptions are classified into two categories: "technical" and "nontechnical." The technical assumptions are made to ensure that every primitive function is "bottom preserving," i.e., it returns bottom if it is applied to bottom. This property ensures that all our functions propagate the error situations denoted by bottom.

A nontechnical assumption represents an assumption that has some consequence on the structure of the design at a lower level. For example, the function majority_of, which is used to define the majority value computed by the majority block, is assumed to satisfy the "commutative property" formalized below. This assumption constrains the behavior of Majority when a majority does not exist for the inputs and requires that the majority value computed in such situations must be independent of the order in which the inputs are presented at the input ports. One way of realizing such a behavior is to compute the majority by taking bit-wise majority of the inputs. The totality assumption on the opcode function imposes the requirement that, no matter what representation we choose to denote the opcode of an instruction, every possible bit pattern of the opcode field must denote some valid instruction.

```
|| Here's what we are assuming about the type data:
data ::= nodata | dataerror | DATUM !number
number ::= Word byte byte byte byte
```

```
opcodeof :: data -> opcode
msbof :: data -> BOOL
majority_of :: data -> data -> data

|| Nontechnical assumptions
AXIOM 'majority_of x x z'='x' , '!z'='True'
AXIOM 'majority_of x z x'='x' , '!z'='True'
AXIOM 'majority_of z x x'='x' , '!z'='True'
AXIOM "majority_commutes" 'majority_of x y z'='majority_of y z x'

AXIOM  "opcode is total" '!(opcodeof x)'='!x'

||Technical assumptions
AXIOM 'msb bottom' = 'bottom'
```

11 Concluding Remarks

The technical contributions of this work are in three areas: *design*, *formal methods*, and *tool building*. In the area of design, we investigated the issues involved in incorporating support for interactive consistency as part of the instruction repertoire of a microprocessor. We designed a communication architecture and the voter hardware necessary for implementing a scheme to achieve interactive consistency among the processors in a four-processor system. We designed the interactive consistency hardware to run under the control of a special instruction on the processors. In the area of formal methods, we developed a method for modeling the design and the correctness of such a byzantine resilient system. We formally specified the system as a multilevel hierarchy and verified the correctness up to a certain level.

The most lasting contribution of our work is, perhaps, the verification technology that has been developed for verifying such systems. To make this technology reusable, we invested a substantial part of our effort in embedding the technology inside Spectool. One significant improvement to the tool was the addition of facilities to support multi-level hierarchy in the design.

Of the one man year of effort devoted to the entire task, we devoted about five man months to upgrading the tool. We spent about the same amount of time to study the interactive consistency algorithm and work out the initial design of the system. We spent the remaining two man months specifying and verifying the design using Spectool. We explored three versions of the voter and made several iterations through the specification-design-verification cycle before we settled on the final version of the design. It is unlikely that this could have been accomplished in two months without the aid of Spectool.

There are three directions in which the present work can be extended. In the horizontal direction, better alternatives to the current design can be explored and verified. As described in section 5.2, several additional properties should be proved about the hardware to show that an appropriate program executing on the microprocessor system reliably accomplishes interactive consistency. The current design requires that all the nonfaulty processors be synchronized (in their program execution) so that they all initiate interactive consistency in the same cycle. It is not possible, in the present design, to make the processors communicate by way of messages to come to an agreement to start the interactive consistency operation. Providing support for such a feature is essential in reconfigurable systems to isolate a faulty processor or to bring into the network a faulty processor that has been resuscitated. Now that a framework to formally analyze such designs has been developed, exploring alternative designs to provably meet other requirements should require less effort.

In the upward direction, we can explore how to formally connect a piece of software expected to run on our computer system with our design. This connection will involve addressing the correctness of the operating systems software that links the hardware to the applications software.

Finally, in the downward direction, we can explore how to rigorously refine our design (or a part of it) into lower levels (gate and switch) of hardware designs. This refinement is a prerequisite to reliably translate our designs into silicon.

References

[1] William R. Bevier and William D. Young. "Machine Checked Proofs of the Design and Implementation of a Fault-Tolerant Circuit". NASA Contractor Report 182099, NASA Langley Research Center, Hampton, VA 23665-5225, November 1990. Authors' affiliation: Computational Logic, Inc., Austin, Texas.

[2] M. Bickford, C. Mills, and E.A. Schneider. "Clio: An Applicative Language-Based Verification System". Technical Report TR 89-13, ORA Corporation, 301A Dates Drive, Ithaca, NY 14850, September 1989.

[3] Mark Bickford and Mandayam Srivas. "Verification of the FtCayuga Fault-Tolerant Microprocessor System Volume 2: Formal Specification and Correctness Theorems". NASA Contractor Report 187574, NASA Langley Research Center, Hampton, VA 23665-5225, 1991. Authors' affiliation: ORA Corporation, Ithaca, NY.

[4] Avra Cohn. "Correctness Properties of the Viper Block Model: The Second Level". Technical Report 134, Computer Laboratory, University of Cambridge, Cambridge, U.K., May 1988.

[5] Warren A. Hunt. "FM8501: A Verified Microprocessor". In *IFIP WG 10.2 Workshop, From HDL to Guaranteed Correct Circuit Designs*, pages 85–114. North-Holland Publishing Co., 1986.

[6] R. Shostak, M. Pease, and L. Lamport. "Reaching Agreement in the Presence of Faults". *JACM*, 27(2):228–234, April 1980.

[7] Mandayam Srivas. "Bridging the Formal Methods Gap: A Computer-Aided Verification Tool for Hardware Designs". In *COMPCON91*, San Francisco, CA, February 25-28 1991.

[8] Mandayam Srivas and Mark Bickford. "Formal Verification of a Pipelined Microprocessor". *IEEE Software*, September 1990.

[9] D.A. Turner. "An overview of Miranda". *ACM SIGPLAN Notices*, 21(12):158–166, December 1986.

[10] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. "Formal Design and Verification of a Reliable Computing Platform for Real-Time Control Phase 1 Results". NASA Technical Memorandum 102716, NASA, Langley Research Center, Hampton, Virginia 23665-5225, October 1990.

A General Structure of a Design Specification

A design specification generated by Spectool consists of the parts described in the following sections.

A.1 Structural Specification Part

This part specifies the data path architecture. It defines a set of types to model the data path state and a set of functions to specify the connections between the data path components. Spectool generates this part in a generic fashion from the following structural information about the data path provided by the user:

1. the names of components and component classes,
2. the names of the actions defined on components,
3. the types that model the internal states of components,
4. the types of the external inputs to the circuit, and
5. the graphical connections between components.

To give the reader an idea of this part of a design specification, a fragment of the structural specification part of the voter circuit is shown below.

```
|| Generic part of the data path state definition
type SYSTEM_STATE = COMP->LOCAL_STATE
type INPUT_STREAM = NAT->EXTSTATE
type CHANGE = <<COMP,NAT,LOCAL_STATE>>
type STATE = <<SYSTEM_STATE, [CHANGE], INPUT_STREAM>>

|| A type for every component class. The type defines the names of
|| all the components used in the data path that belong to the class.
majority ::= MAJ1 | MAJ2 | MAJ3
bytereg  ::= PR1 | PR2 | PR3 | R12 | R13 | R23 | R31 | R32
bitlatch ::= VS | ST
```

```
|| A labeled union type of all the component class types
COMP ::=   Controller | C_majority !majority | C_mux4 !mux4
         | C_bytereg !bytereg | C_bitlatch !bitlatch

|| A labeled union type of all local states of components
LOCAL_STATE ::=   S_Controller !CONTROLSTATE
                | S_majority !majority_localstate
                | S_mux4 !mux4_localstate
                | S_bytereg !bytereg_localstate
                | S_bitlatch !bitlatch_localstate

ACTION ::=   A_majority !majority_ACT
           | A_mux4 !mux4_ACT
           | A_bytereg !bytereg_ACT
           | A_bitlatch !bitlatch_ACT

|| The following specifies the input connections to the component MAJ2
majorityinput s (C_majority MAJ2) =
   <<getbyteregout0 (current s (C_bytereg R32)),
     getbyteregout0 (current s (C_bytereg R12)),
     getbyteregout0 (current s (C_bytereg PR2))>>
....
```

The type STATE defines the data path state as a tuple: <<SYSTEM_STATE, [CHANGE], INPUT_STREAM>>. SYSTEM_STATE maps every component (an element of type COMP) in the data path to its LOCAL_STATE. [CHANGE] is the list of *pending* actions on the data path components. This list is maintained to simulate the effect of delays on actions. The list contains an update record for each of the actions that has been triggered by the controller on components, but is yet to be completed. The first two fields of STATE define a snapshot in time of the data path state. The state of a component is given by SYSTEM_STATE unless an action is pending on the component, in which case the state is bottom.

The third field of the tuple, INPUT_STREAM, specifies the values of the external inputs to the circuit as function of time. INPUT_STREAM is a function type from time (NAT) to EXTSTATE, where EXTSTATE is a type that combines all the external inputs into a tuple.

The type LOCAL_STATE is a labeled union of the *local states* of all the components of the data path (and the controller). The local state type of a component is a tuple of its internal state and (a tuple of) its outputs. The definition of the local state type of a component appears as part of the component class specification.

The set of components in a data path is organized so that every component is an instance of a component class. The type COMP groups together the names of all the components in a data path. It is defined as a labeled union of all the component class types, where a component class type is an enumerated type of the names of all the components belonging to the corresponding class.

Similarly, the type ACTION groups together the names of the It is defined as a labeled union of all the *action types* of the component classes, where the action type of a component class is an enumerated type of the names of all the actions defined for the class. The definition of the action type of a component appears as part of the component class specification.

Connections between data path components are specified by defining, for every component, a function that determines the inputs to the component in a given data path state.

A.2 Component Classes Specification Part

Every component in the data path is an instance of a component class. The components belonging to the same class share several attributes. This part specifies the shared attributes of a class for every class used in a design. A component class specification defines the type that denotes the internal state of a components of the class, the types of the outputs, a type denoting the names of the actions on the components, and the effects of the actions on the components. For every action, it defines three functions: "state" and "output" functions return, respectively, the new state and the new outputs of a component after an action is performed; and the "delay" function gives the delay associated with the action. For illustration, a part of the specification of the majority component class used in the voter circuit is given below.

```
|| The component class majority.
|| Local state of majority: <<internal state type, <<output types>>>>
type majority_localstate = <<<<data, BOOL>>,<<(byte),(BOOL)>>>>
```

```
|| Type denoting the actions on the components of the class
majority_ACT ::=
 select3 !majority |
 select2 !majority |
 select0 !majority |
 select1 !majority |
 comp !majority

majoritydelay (select3 c) = #0
majoritycomp (select3 c) = C_majority c
majorityout (select3 c) s <<in1,in2,in3>> = <<get_byte 3 (dataof s), bitof s>>
majoritystate (select3 c) s <<in1,in2,in3>> = s

majoritydelay (select2 c) = #0
majoritycomp (select2 c) = C_majority c
majorityout (select2 c) s <<in1,in2,in3>> = <<get_byte 2 (dataof s), bitof s>>
majoritystate (select2 c) s <<in1,in2,in3>> = s
....
```

A.3 Controller Specification Part

A controller is specified by means of two functions `nextstate` and `scheduler`. The `nextstate` function gives the next controller state as a function of the current state and controller inputs. The `nextstate` function is used at the end of each cycle to advance the controller state. The function `scheduler` returns a list of actions as a function of the controller state, controller inputs, and the phase. For illustration, a part of the specification of the controller for the voter circuit is shown below.

```
CONTROLSTATE ::=  LDP1 | LDP2 | XNG11 | XNG12 | XNG21 | XNG22 | XNG31
                | XNG32 | CMPP | OUT1 | OUT2

nextstate LDP1 in = (~(startedof in))->(LDP1);(LDP2)
nextstate LDP2 in = XNG11
...

scheduler XNG11 <<vstart, started>> 0 = (selprvt1byte0)
```

```
selprvt1byte0 = [A_bytereg(read0 PR1), A_mux4(choose3 MUX)]

scheduler XNG12 <<vstart, started>> 0 = (selprvt1byte2)
selprvt1byte2 = [A_bytereg(read2 PR1), A_mux4(choose3 MUX)]
...
|| Some of the details specific to the voter design
|| There are 4 clock phases per cycle.
num_phases = 4
```

A.4 Composite Behavior Specification Part

This part defines a set of functions that derive the composite behavior of a design using the information expressed in the rest of the specification. This part formalizes in Caliban the operational model of the behavior of a finite state controller system. This part is identical for every circuit since Spectool generates it in a completely generic fashion. The higher-order function definition capability of Caliban is a primary reason why it is possible to express this part in a generic fashion. A top level fragment of this part of the specification is shown below.

```
Execute s = do_phases 0 s

Output s = generate_output 0 s

do_phases n s = update_state s , n = num_phases
                do_phases (n+1) (do_phase n s)

do_phase n <<s,p,in>> =
  advance_inputstream (do_actions (current_schedule s2 n) s2)
  where s2 = update_state <<s,p,in>>
```

The two main functions defined in this part are `Execute` and `Output`. `Execute` advances the state of the system across a single cycle. `Output` returns the (tuple of) external outputs produced by the circuit over the next cycle. `Output` actually returns a list outputs, one for every phase in a cycle. The two functions are defined hierarchically in terms of several other functions, some of which are described below.

To simulate the effect of delays on component actions, the specification maintains a list of records of pending updates that have been triggered by the controller, but are yet to take effect on the circuit state. The update record for an action contains the component on which the update is pending, a time-out counter that is initialized to the delay of the action and decremented after every phase, and the result of the update. The function `update_state` causes the result of all the update records on the list that have timed out to take effect on the circuit state. The function also decrements the time-out counter of the update records that have not yet timed-out by one unit of time. The purposes of the rest of the functions should be apparent from the names of the functions. They are summarized below.

- The function `do_phases` updates the state for all the phases in a cycle.
- The function `do_phase` updates the state for a single phase. It causes (using `update_state`) the pending updates that have timed out to take effect; gets the `current_schedule` from the controller, makes new update records (using `do_actions`) for all the actions in the schedule, and then advances the input stream by a time unit.

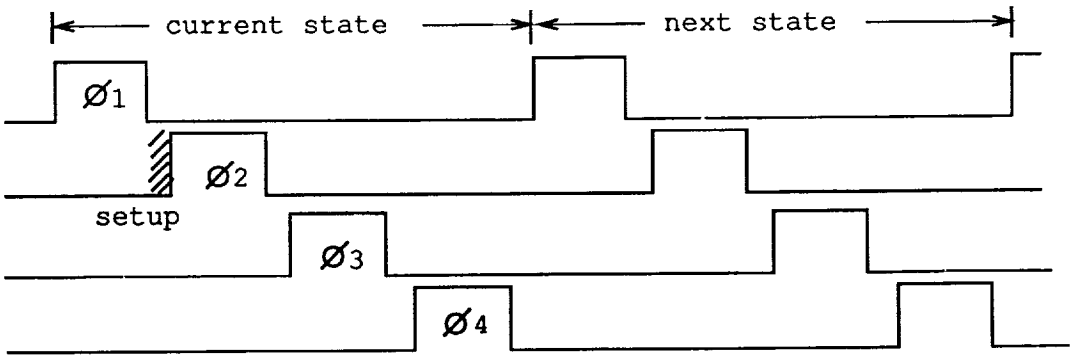


Figure 7: Clock Timing Convention

B Timing Constraints of the Hardware Model

B.1 Timing Implications to Lower Level Design

Figure 7 illustrates a connection between our abstract timing model and a more concrete one with an explicit clock with four non-overlapping phases. We use the convention that the actions scheduled in a certain phase of a state are triggered by the active (rising) edge that immediately follows the phase. The action `nextstate`, which moves the controller from the present state to the next, always occurs in phase 4. (Note that the `nextstate` action must have a unit delay.)

The constraints that our timing model imposes on a lower level hardware design where the clock signal is manipulated explicitly are the following. The system clock edge must reach each element in the circuit simultaneously. In practice, every clocked component requires that its inputs be stable for a certain period ("setup time") prior to the clock edge. The setup time is not explicitly taken into account in our timing model. But, our timing analysis ensures that the setup time constraint is met provided every signal changes only synchronously with the clock and the "zero" delay actions are handled properly at the lower level. One must ensure that the real expected delay (δ) of the logic circuit that implements a "zero" delay action is small enough so that "phase time," i.e., the time between two consecutive active edges, must be greater than (δ + setup time).

The last constraint, however, does not imply that our design model precludes the use of a combinational block that is expected to have a significant amount of delay relative to a given clock period. The constraint means that the action implemented by the block must

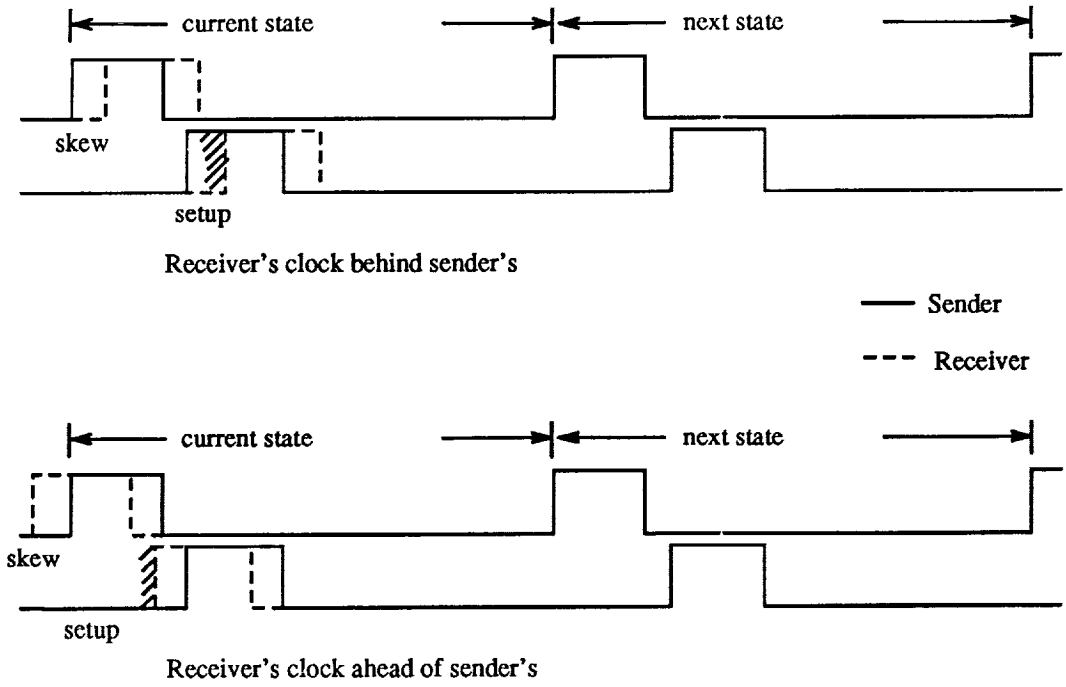


Figure 8: Implications of Clock Skews

be assigned a delay (in phases) that is at least as large as the expected delay (in real time). Then, our timing analysis will ensure that the outputs of a combinational block will not be used before it stabilizes.

B.2 Timing Implications of Clock Skew on Design

Given a concrete elaboration (as shown in Figure 7) of our timing model in terms of clocks, we can analyze some of the effects of a nonzero skew between the corresponding active edges of the clock signals controlling two communicating blocks in a design. Suppose an action scheduled on a receiver in a certain phase of the clock needs to use a signal that is expected from a sender in the same phase. If the clocks of the receiver and sender are perfectly synchronized (zero clock skew), then the sender's signal will stabilize in time to satisfy the signal setup requirement of the receiver. However, suppose there is a skew between the two clocks. Figure 8 shows the two possible situations. If the receiver's clock is behind sender's, then to meet the setup requirement for the receiver, the sender's signal must not change for at least as long as the clock skew period in the following phase. If the receiver's clock is ahead of the sender's clock, then the clock skew must be small enough to allow for the signals produced by

the sender to stabilize before the receiver's setup time. If the signals stabilize instantaneously in the phase in which they are produced, then there will not be a problem. Since there will always be some real combinational delays (for zero delay actions), this situation imposes the following condition: *clock skew* $\leq$ (*phase time* $-$ (*set up time* $+$ *combinational delay*), where *phase time* is the time between two consecutive active edges. The first constraint can be satisfied by choosing a large clock period relative to the worst case clock skew. We ensure that the second constraint is not violated by using the following safe design convention: *The sender must hold its signal for one additional phase following the phase in which the receiver is supposed to use the signal.*

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-4381		

4. Title and Subtitle

Verification of the FtCayuga Fault-Tolerant Microprocessor System

Volume 1: A Case Study in Theorem Prover-Based Verification

5. Report Date

July 1991

6. Performing Organization Code

7. Author(s)

Mandayam Srivas and Mark Bickford

8. Performing Organization Report No.

10. Work Unit No.

505-64-10-05

9. Performing Organization Name and Address

ORA Corporation
301A Harris B. Dates Drive
Ithaca, NY 14850-1313

11. Contract or Grant No.

NAS1-18972

13. Type of Report and Period Covered

Contractor Report

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes

Technical Monitor: Ricky W. Butler, Langley Research Center
Task 1 Final Report
Volume 2 published as NASA CR-187574.

16. Abstract

This paper presents a formal specification and verification of a property of a quadruplicately redundant fault-tolerant microprocessor circuit design. The circuit performs the task of attaining interactive consistency among the processors using a special instruction on the processors. The design is based on an algorithm proposed by Pease, Shostak and Lamport. The property verified ensures that an execution of the special instruction by the processors correctly accomplishes interactive consistency, provided certain preconditions hold. An assumption is made that the processors execute synchronously. The verification was done using a computer-aided hardware design verification tool, Spectool, and the theorem prover, Clio, both of which were developed at ORA. A major contribution of the work is the demonstration of a significant fault-tolerant hardware design that is mechanically verified by a theorem prover. The work illustrates the advantage of using hierarchy and abstraction in system design specification to manage the complexity of formal verification. It demonstrates the value of a special-purpose tool that tailors the use of a theorem prover to a hardware domain in order to reduce the effort required for formal verification.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Fault Tolerance Formal Verification Byzantine Agreement Hardware Verification Mechanical Theorem Proving	Unclassified - Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	61	A04

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171

NASA-Langley, 1991